AD-A048 053 STANFORD UNIV CALIF DIGITAL SYSTEMS LAB F/G 9/2
COMPUTING IN STORE.(U)
JUN 77 J K ILIFFE
UNCLASSIFIED DSL-TN-117 N00014-75-C-0601
NL

1 OF 1
AD A048053

END
DATE FILMED
1 -78
DDC

(11)

FG.

COMPUTING IN STORE

by

John K. Iliffe

(See back page for 1473)

June 1977

Technical Note No. 117

Digital Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 94305

The work described herein was supported in part by the Joint Services Electronics Program under Contract No. N00014-75-0601.

Digital Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 94305

Technical Note No. 117

(11) June 1977

(6) COMPUTING IN STORE

by

(12) (10) John K. Iliffe

ABSTRACT

These notes provide an introduction to the class of single-instruction, multiple-data stream computers with the simplest processing elements. Design principles are explained in terms of hypothetical Distributed Processor Arrays, with examples drawn from experimental systems. Emphasis is placed on (a) minimising the cost differential when the DPA is compared with conventional main storage, and (b) designing the array control unit to support advanced forms of protection and language implementation. The influence of the DPA on general system design is examined briefly.

The work described herein was supported in part by the Joint Services Electronics Program under Contract No. N00014-75-0601.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 DISTRIBUTED PROCESSOR ARRAYS

These lectures are concerned with assemblies of processors, each having local arithmetic and local storage capability but sharing a common control unit, so that they execute synchronously a broadcast stream of instructions. When good (in terms of performance) such arrays can be expected to be very, very good, but when they are bad they are unproductive if not exactly horrid. Thus one of the main objects of system design is to keep the array working on problems at the favourable end of the spectrum for a sufficient proportion of time to justify its cost. In the Illiac IV machine an attempt has been made to achieve that end by covering an extensive user catchment area by using a communications network, enabling remote sites to send problems to the array: there are very few laboratories or businesses that present a continuous workload with the high degree of parallelism necessary for efficient working. An alternative approach is to reduce the cost attributable to the array to the extent that it can operate economically at a low duty cycle. In principle, one would like to see a T% improvement in system throughput for an investment of substantially less than T%, but in practice we shall see that the presence of an array can affect system design in ways which are impossible to quantify.

Readers unacquainted with array processor designs will gain some insight from early papers on Solomon [1] and Illiac IV [2]. One of the main performance bottlenecks on conventional systems is the data path from processor (or processors) to program storage units: with random access to words the fastest crossbar or bus systems achieve peak rates of about 10^8 bytes/second. There are two methods of exceeding this limit, both of which assume non-random access patterns:

(a) Use vector addressing modes, which allow retrieval and storage of word sequences regularly spaced in store. It is then necessary to transmit only a fraction of the addresses, and speed

gains can be achieved by interleaving store cycles. The CDC Star, TI ASC and Cray-1 machines provide examples of this approach which leads to maximum data rates in the region of 10^9 bytes/second.

(b) Use a new pattern of physical interconnection in which each processor has direct access to only a limited region of storage. Thus Illiac IV with 64 local stores containing 8 byte words and cycling at 240nsec would achieve a maximum data rate of about $2*10^9$ bytes/second. It will be seen later that practical data processing rates exceeding 10^{10} bytes/second can be envisaged with currently available technology, at the expense of severe limitations on accessibility.

Of course, maximum data rate is not the end of the story, and although it is true that data access patterns are non-random they vary appreciably from one class of problem to another, allowing various degrading factors to come into play. We should note in passing that the class of problems of immediate interest are charcterised by regular data spacing, which is not well served by slave memory techniques. On the other hand slave stores deal with the type of non-randomness which appears as repeated reference to the same locality, so the two mechanisms are not competing for the same class of problem and we may hope to see them working effectively in combination in some future system.

For practical purposes only linear or rectangular arrays need be considered. Handling three dimensional arrays is severely limited by the planar form of hardware, which is unlikely to change until radically new methods of manufacture are proven. The form of interconnection within a plane is more open to debate. Many problems are naturally expressed in polar form, or by using a hexagonal cell pattern rather than square; the result of mapping them into a rectangular array is to leave some of the processors and connection paths unused. However, in the light of experience gained so far the square array with four near-neighbour connections appears to be most widely applicable.

The main application incentive derives from the numerical solution of field equations such as those occurring in reactor physics and meteor-

ology, in which method (b) is not unduly restrictive. Moreover, most numerical methods are based on a discrete representation of physical space and time that can be mapped directly onto the array, the local storage providing the third and fourth dimensions when necessary. Problems of this class create a practically unlimited demand for computing power in the quest for high resolution and there is no doubt that very high absolute performance and high performance/cost are attainable using array techniques. The principles of such applications are outlined below, but the emphasis of discussion is on non-numerical problems, system and language design.

The main engineering stimulus comes from the emergence of semiconductor stores as main memory components: having the same physical and electrical properties as logical devices it is far easier to consider closely-coupled assemblies than in the days of core memory. Possible applications of this principle to cache memories have been pointed out by Stone [3]. A related factor of extreme importance is that simple and highly repetitive circuits are very suitable for LSI manufacture: some of the processors to be considered require less than 100 logic gates each and could eventually represent a negligible cost increase. Simplicity is the consequence of using single-bit wide data paths and providing a primitive instruction set. The complex functions that are needed for arithmetic and data manipulation reside in the form of stored program in the same way as microcode for a conventional machine. It is possible that reduced hardware costs will allow us to regard a processor of 1000 gates as 'negligible' at some future date, at which time commitment to greater functionality or wider data paths can be considered as an alternative to closer packing of single bit processors. The main requirement at present, however, is to understand the trade-offs well enough to make sensible decisions, and the study of single-bit processors seems to be the best starting point for that.

The reader may recognise the resemblance to cellular arrays [4], [5], whose study is prompted by the same technological projections. The main difference is that the logic and data paths are thought of as fixed in the processor array and variable (often on a row or column basis) in many cellular designs. There is no intrinsic reason for maintaining the distinction. Further research may show effective ways of combining the two lines of development.

1.1 DPA storage

A store module can be thought of as a rectangular array of (semiconductor) storage elements whose dimensions are determined as a multiple of the store data word size and a power of two. For illustration a 'toy' store of 16-bit words in 16 rows will be used and I shall denote by 'DPAn' where n is in the range 1 to 9 a square array with sides of length 2^n. Thus the toy array is referred to as DPA4. The practically useful arrays in terms of computing power require n at least 6. Parity and/or tag bits are assumed to be present above the nominal word size, but they do not take part in the array processing activities and they will be omitted from the following description. Figure 1 shows a plan view of the store module and a '3D' view with the storage bits extended in the vertical direction to show the layout of words in horizontal sections through each row of storage elements. Each storage element contains a few thousand binary digits: I shall refer to an array with m kbit stores as 'DPAn.m'. Thus the toy array, which has 1024 bits in each element, is DPA4.1

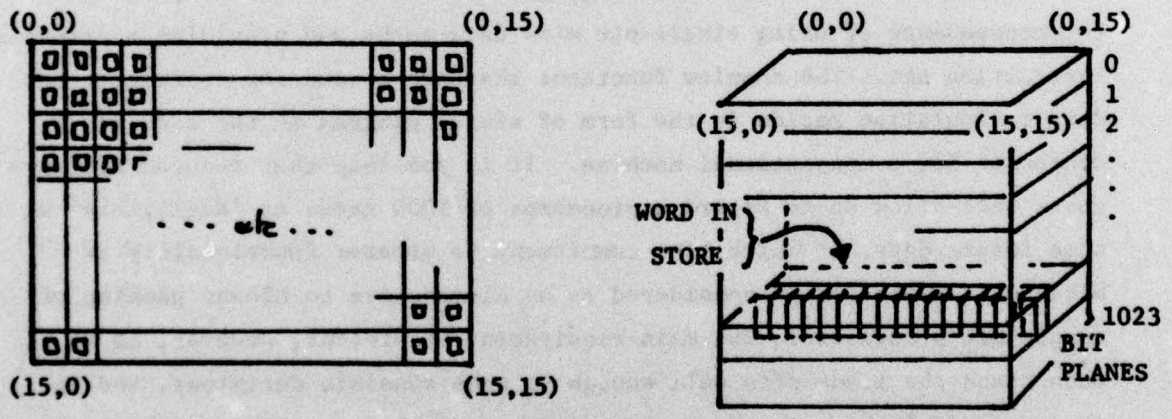


Figure 1: Storage module layout for DPA4.1

The array processor attaches a small computer or processing element(PE) to each storage element. It is in that sense that processing power is 'distributed' through the store. The control logic normally associated with the store module for handling addresses and data is elaborated to form instructions that are broadcast in synchronism to every PE

in the array: each PE must obey the instruction, the only option that can be exercised locally is whether to store the result. Figure 2 shows a plan of the PE array and a perspective view with the processors in the lid of the store. In programming terms we shall see that data is processed by passing it 'vertically' through the lid of the store 2^{2n} bits at a time from a selected bit plane.

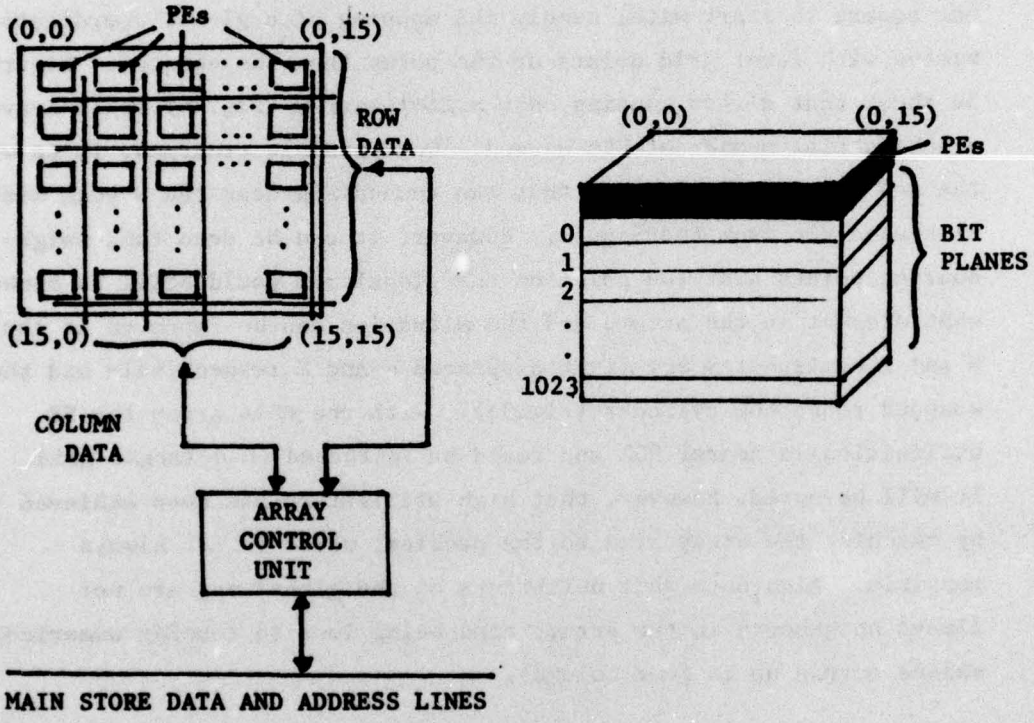


Figure 2: Storage module with processing elements DPA4.1

Each PE has single digit data connection to neighbours in four directions designated N, E, S, and W. Elements at the edges of the array are always short of one or two neighbours. The input at these points can be selected by program to be (a) always zero; (b) taken from the other end of the same row or column to give cylindrical or toroidal geometry; (c) taken from the other end of the next row (or column) to form a linear or circular array of 2^{2n} PEs. In each geometry the thickness of the surface or line is determined by the local store size, eg 1024 bits in the toy machine.

Example

The choice of PE connections is a compromise between pin limitations inherent in the form of construction and problem requirements that have become apparent in application studies. The consequence of omitting a connection is that data has to be routed 1 or 2^n cells at a time across the array for as many bits as there are in the data. The result can best be illustrated by considering a problem that is not square to start with, namely the mapping of a global coordinate system with fewer grid points at the poles than the equator. Figure 3a shows that direct mapping onto a 24*16 array (Fig.3b) would leave a substantial number of PEs unused. One possible treatment is to wrap the map round a cylinder so that the unused PEs near the N pole mesh with used PEs from the S pole. However, it can be seen that neighbouring points near the poles on the global map would still be somewhat distant in the array, and the situation can be improved if the N and S hemispheres are first displaced W and E respectively and then wrapped round the cylinder (Fig.3c). With the DPA4 array the PE utilisation is nearer 90% and could be increased on a larger grid. It will be noted, however, that high utilisation has been achieved by matching the array size to the problem, which is not always possible. Also note that neighbours on the global map are not always neighbours in the array, time being lost in routing numerical values across up to four columns.

We can readily identify three factors which prevent the DPA from achieving the theoretical data processing rate of 2^{2n} bits per instruction:

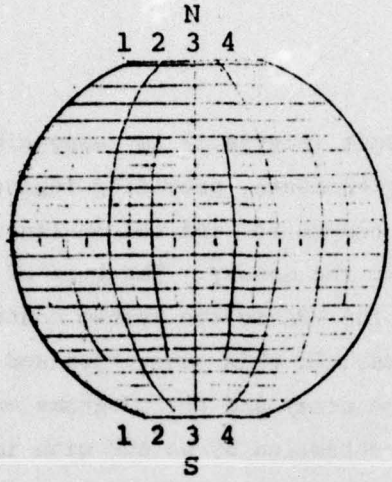
(a) mapping which prevents a problem from being cast into a form to use all PEs;

(b) routing which occupies the array in unproductive data movements;

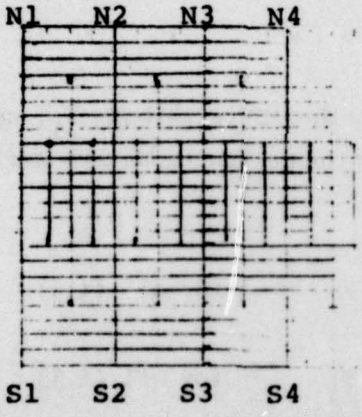
(c) branching which causes only a fraction of the PEs to be active during a particular phase of calculation, eg in the preceding example the poles may require special calculations which could use only $\frac{8}{256}$ of the available PEs.

Experience shows that intuitive judgements based on habits formed in using conventional machines can be quite far from the truth, particularly in the area of (a) and (b).

(a)

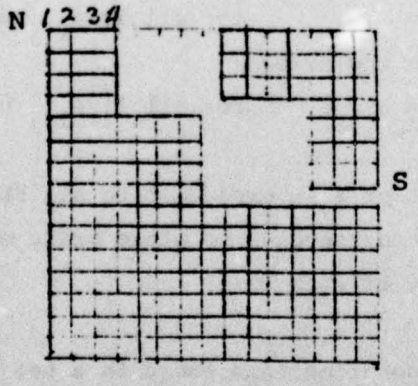


(b)



$\frac{224}{384}$ = 58% used

(c)



$\frac{224}{256}$ = 88% used

Figure 3: Representing a Spherical Map

1.2 Array operations

The Array Control Unit (Figure 2) is responsible for broadcasting instructions to the processing elements, providing registers to buffer the data sent and received on the common row and column lines, and for serving external requests for access to the memory. The type of external request depends on the part played by the DPA in the system context, which I shall return to in the later lectures. In this subsection and the next we consider the elementary functions of the array and the programs executed by the ACU. It is assumed that a DPAn is controlled by an ACU with internal data width 2^n, eg the DPA4 is associated with 16 bit data fields in the ACU, which are placed in correspondence with the row and colomn data lines: otherwise the description would have to provide operations to align part-words with the array edges, or conversely.

Let Y be a register in the ACU with bits Y_i, and let $A_{i,j}$ be a bit in the (i,j)th PE. Then four input functions are defined as follows:

(1) Input by row:

$$A_{i,j} = Y_i \qquad \text{for all } (i,j)$$

(2) Input by column:

$$A_{i,j} = Y_j \qquad \text{for all } (i,j)$$

(3) Input by row with column select:

$$A_{i,J} = Y_i \qquad \text{for all } i;\ A_{i,j} \text{ unchanged if } j \neq J$$

(4) Input by column with row select:

$$A_{I,j} = Y_j \qquad \text{for all } j;\ A_{i,j} \text{ unchanged if } i \neq I$$

Thus in cases (1) and (2) a bit of Y is broadcast to all PEs along a row or column of the array. Case (4) corresponds to store write when the selected bit $A_{i,j}$ is in the local memory of each PE.

Corresponding to input functions there is a set of four output functions, with and without selection by row or column. Here the resultant bit in the Y register is the logical and of all selected PEs on the data line:

(1)' Output by row:

$$Y_i = \bigwedge_j A_{i,j} \qquad \text{for all } i$$

The output operation 'by column with row select' corresponds to normal store read when the $A_{i,j}$ is in local memory. The method of choosing the A and Y words is discussed later.

In its simplest form the PE contains three single-bit registers with the following uses:

A is the activity register. When zero, writing to local stores can be inhibited;

B is the arithmetic and logical accumulator;

C is the carry digit.

The other components of the PE are a routing multiplexor, which is used to select input to local store from the A or B registers, near neighbours (N, E, S, W), or the common row (R) or column(C) data lines, and the local memory itself. An inverter (I) allows the polarity of data to be reversed in going

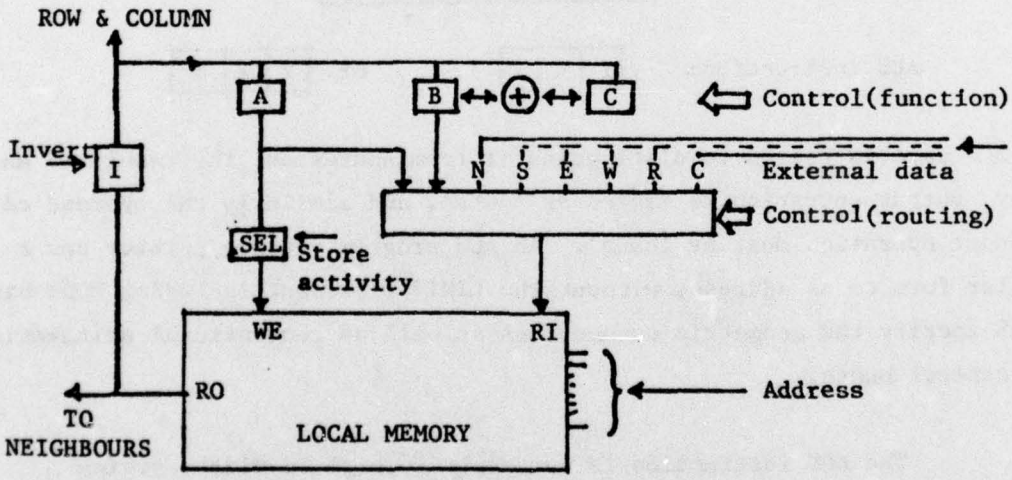
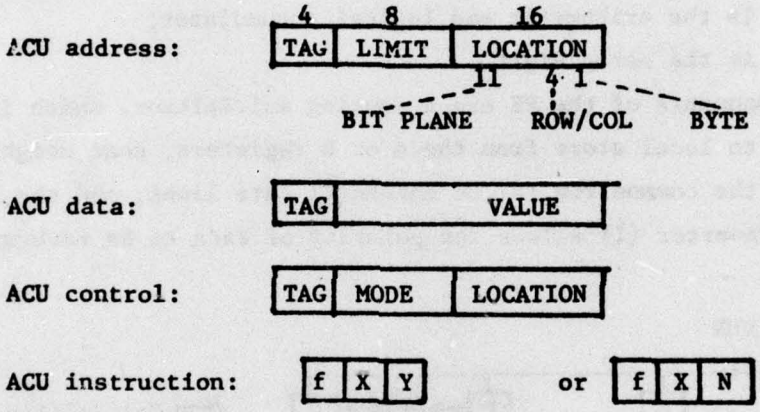


Figure 4: Processing element schematic for the DPA

from memory into the A or B registers, and the or gate SEL allows program control of the use of A to inhibit 'store' operations.

The PE carries out two types of operation: functions of the arithmetic registers A,B and C; and routing of data. Each uses the broadcast local memory address, which is the same for all PEs, so that the operation is carried out on a 'bit plane' in the DPA. An address is specified by an ACU register which for security reasons also contains a limit field giving the maximum modifiable range of that address. To address the DPA4 we take a 16 bit byte location, a 12 bit limit and a 4 bit tag field. The most significant 11 bits of the location select a bit plane, the remainder specify column or row select when necessary (4 bits) and the least significant bit selects even or odd byte.



The ACU data is tagged to distinguish it from addresses, the result of an array output operation is tagged as 'data', and similarly the operand of an input operation must be 'data'. An ACU program control pointer has a similar form to an address, without the LIMIT field but including MODE bits which specify the geometric connections as well as conventional arithmetic and control modes.

The ACU instruction is uniformly 16 bits in width, giving a function field f and either two 4-bit register addresses X, Y or a single register and a literal field N. The instruction set covers the requirements of sequential operations in the ACU itself and parallel operations in the DPA of the two types mentioned above.

DPA ARITHMETIC

The X-register address is used to select a local data value x in each PE. The following functions are available:

LDA	Load A	Sets A = x
LDB,LDC	Load B,C	Set B = x, C = x respectively
ADD	Add to acc.	Forms the sum of B,x and C in B and forms the carry in C
AND	AND to acc.	Forms the logical and of B and x in B
OR	OR to acc.	Forms the logical or of B and x in B
EQU	EQU to acc.	Forms the logical equivalence of B and x in B

In any arithmetic function the datum can optionally be inverted.

DPA ROUTING

The X-register address is used to select a destination plane or in some cases the source. The following functions are available:

IR	Input by row	Uses Y to provide data input to x according to (1) on page 10
IC	Input by column	See (2) on page 10
IRC	Input by row with column select	(See (3))
ICR	Input by column with row select	(See (4))
AOR	AND output by row	(See (1)')
AOC	AND output by column	(2)'
ORC	Output by row with column select	(3)'
OCR	Output by column with row select	(4)'

Note that ICR corresponds to conventional STORE and OCR to LOAD functions.

MVN	Move North	The datum plane is moved north one PE
MVS, MVE, MVW		Similarly for south,east and west.

Note that MVE, MVW correspond to single bit word shifts in the array, the 'most significant' or 'left' end of a word assumed to be on the W edge.

STA	Store A	Sets x = A
STB	Store B	Sets x = B

In any routing function the store action is by default conditional on the value of A = 1 in the destination PE; it is possible to override A in any instruction by following the function with "/U" as in:

eg STB/U Store B unconditional , ie independent of the value of A

1.3 The array control unit

The ACU obeys instructions fetched from the DPA. Its operations include those listed in the previous subsection together with conventional control, arithmetic, logical and addressing functions. The ACU is classed as a Pointer-Number machine, ie one in which a distinction is drawn between pointers (control and data addresses, codewords or capabilities) and numbers. Functions are provided to create and manipulate various classes of objects, the instruction set being designed to prevent abuse in the sense of damaging integrity or gaining access to objects without permission. However, only one aspect of the overall design need concern us here and that is the use of addresses to refer to bytes, words or bit planes in the DPA.

The form of data address is given on p.12. The 12-bit limit field enables an address to refer to up to 128 consecutive bit planes. The low order 5 location bits are used in byte and word access, including row and column selection. To simplify the addressing rules when working with array data we declare that protection of binary segments is only resolved to the bit plane boundaries.

An ACU program will be written as a sequence of statements with elementary IF, GOTO, WHILE and DO control clauses. Functions of the ALU are expressed by the following operators:

ARITHMETIC AND LOGIC

syntax	symbol	function	operand types	example
binary infix	+	add	N, N	x + y
binary infix	-	subtract	N, N	x - y
unary prefix	-	negate	N	-z
binary infix	*	integer mpy	N, N	4 * p
binary infix	&	logical and	N, N	x & #FF
binary infix	/	logical or	N, N	x / z
binary infix	%	logical neq	N, N	a % b
binary infix	<<	left shift	N, N	p << 3
binary infix	>>	right shift	N, N	a >> b

ADDRESSING

binary infix	'	modification	A, N	a ' 8
binary infix	↑	limitation	A, N	b ↑ n
unary postfix	.	load	A	x.

ASSIGNMENT

binary infix	=	register transfer	-,-	x = y
binary infix	=.	store	A,-	x =. y

In each case the operands are ACU registers that will be declared as required, or literals where numeric arguments are allowed. Expressions are evaluated from left to right, addressing operations taking precedence over arithmetic and assignment. Where no assignment operator is present and the first operand is a register the result overwrites the register as in the statement "x'1", which modifes the register x by 1.

The usual conditions are set by arithmetic and logical operations and tested in control clauses (NZ, ZE, GT, GE, LT, LE, OV, NV). The addressing functions produce invalid (null) results in the event of protection violation and set the condition IR with inverse VR (valid result). Arithmetic and control functions fail if an operand of the incorrect type is presented. In most of the examples given below such exceptions are assumed not to occur. The protection rules simply ensure that a program does not cause damage outside the protection domain defined by the ACU registers when a programming error occurs.

DPA functions will be expressed using as prefix or infix operators the mnemonics given in the previous subsection. Arithmetic functions require one argument (an expression giving the address of a bit plane), which will be preceded by "-" when inverting the input to the PE. Store functions require one argument giving the address of a bit plane. Array input requires a destination (bit plane address) and source (data register). Output requires a destination register and source. Finally, move operations require a bit plane address and step count (data).

The example on the following page illustrates the conventions used in writing ACU programs. It is assumed that the ACU supports a procedure calling mechanism so that the function in the example would be called as "SUBTRACT(P, Q)", where P and Q specify operands. The program will abort if Q is longer than P and set OV if overflow occurs anywhere in the array, the common plane OFLOW indicating which elements overflowed. The mechanism of procedure call and module interconnection will be examined in a later lecture because it is affected by the presence of the DPA.

```
/*  Example:
The following program segment subtracts two arrays of (P+1)
bit 2's complement integers stored in vertical form.  The
result ARG1 - ARG2 is stored in RES in all active PEs.  The
ACU OV condition is set if any result overflows.  The boolean
matrix OFLOW is set = 0 wherever overflow has occurred in an
active PE.  The arguments are specified by bit plane addresses
with least significant digits in the  high address plane  */

REGISTERS [ ARG1  ARG2  RES  P  OFLOW  ]

/* Set carry in  and initialise overflow plane */
OFLOW  IR/U  -1 ; LDC OFLOW; P<<5
/* Subtraction loop */
WHILE GE DO (LDB ARG'P; ADD -ARG2'P; STB RES'P; P-32)
/* Set overflow plane to zero wherever overflow has occurred */
LDB ARG1; ADD -ARG2; EQU RES; STB OFLOW;
P AOR OFLOW; IF (P ≠ -1) (SETOV); RETURN
```

2 ELEMENTARY DPA PROCEDURES

Having shown in principle how a DPA is controlled we can examine its application to some frequently occurring tasks. The object is to obtain theoretical performance limits, taking into account PE utilisation, routing and branching. Obviously there is no point in pursuing an application unless it offers substantial returns on that basis.

A fourth degrading factor has to be added to those listed on p.8: the time taken to supply the ACU with instructions and the time taken by the ACU in modifying address counters, testing for loop termination, etc, in which potential array functions are 'lost'. There are several ways of minimising the loss, including instruction buffering and overlap, but I do not propose to discuss them here and shall assume instead a moderate time of DPA execution (200nsec) in which allowance has been made for the effect just mentioned. On that basis the subtraction example given at the end of the first lecture requires:

$$3P + 10$$

DPA cycles, from which it can be seen that if the precision is large (say greater than 20) the 'end effects' are negligible, but if it is small, as is frequently the case, we should be looking for better ways of setting carry and testing overflow. However, let me emphasise the importance of evaluating any such improvement in terms of its contribution to overall system throughput rather than to individual procedures.

The procedures are classified as 'arithmetic and logic', which are mainly concerned with operations within a PE or row of PEs without regard to neighbours; 'routing', which are concerned with preparing arrays for parallel arithmetic; and 'matrix', which combine the first two. The objective of a more complete study would be to provide a set of arithmetic and data manipulative function that can be used by application programmers and compilers in generating array code and, perhaps more valuable in the long run, to develop the intuitive understanding of the array which is essential to successful systems analysis.

2.1 Arithmetic and logical operations

There are two word orientations of importance in DPA operations: horizontal and vertical. The former corresponds to the conventional store layout, so that data written as words by the ACU can be processed in situ by the array. The latter form, which was used in the example of subtraction, requires the data words to be stored in consecutive bit planes, one word to each PE. The DPA4 can process 16 words of 16 bits in horizontal form, or 256 words one bit at a time vertically. The distinction is less important in logical operations, the bit processing rate being the same in either case, than for arithmetic, in which provision has to be made for carry propagation. When dealing with large arrays of short words the vertical form is to be preferred because it allows greater PE utilisation, and in certain special functions such as the manipulation of Boolean arrays or sign digits it is about N ($=2^n$, the word length of the array) times faster than horizontal and N^2 times faster than sequential processing in the ACU. (More precisely, such comparisons should read that in the limit, for large arrays, the ratio is k*N or $k*N^2$ where k is a small constant factor, usually near unity.)

In vertical mode, carry is propagated through the C register in each PE. In horizontal mode, convention requires carries to propagate to the west, which would be so time-consuming in the DPA that it would have little practical use. We shall see later how additional routing and/or function can be used to achieve competitive speeds in horizontal mode. When summing a large number of word planes the DPA is placed at less of a disadvantage by using carry-save techniques. For example, horizontal multiplication in DPA4 requires the summation in each row of PEs of 16 expressions of the form:

$$\sum_{i=0}^{15} b_i^j * 2^i \qquad \text{for } j = 0 \text{ to } 15$$

where b_i^j is the ith bit of the jth word, which occurs in the ith PE. The product can be formed by summing vertically to give the non-standard result:

$$P = \sum_{i=0}^{15} c_i * 2^i \quad \text{where } c_i = \sum_{j=0}^{15} b_i^j$$

Each c_i is a four-bit carry which can be propagated by MVW, followed by summing again vertically. The final addition, which completes the carry propagation, is probably best done in the ACU. (This is one of the appli-

cations in which the end-effects on addition become important.)

The low level of coding allows advantage to be taken of special properties of the data in many instances. Multiplication by a constant, for example, is faster than array multiplication in all cases because it can take advantage of strings of zeros or ones in the multiplier. Iterative calculations such as square root can use a low precision approximation in the early stages. Note also that in squaring operations the coefficient b_i^j which is in fact $b_i.b_{j-i}$, where the b_i are the digits of the operand, also occurs as b_{j-i}^j. Therefore c_i can be computed as:

(i even) $$c_i = 2*(b_i.b_0 + b_{i-1}.b_1 + \ldots + b_{i/2-1}.b_{i/2-1}) + b_{i/2}.b_{i/2}$$

(i odd) $$c_i = 2*(b_i.b_0 + b_{i-1}.b_1 + \ldots + b_{(i-1)/2}.b_{(i+1)/2})$$

which halves the number of partial products.

In general, vertical multiplication of two p-bit numbers requires p additions to give a 2p bit result, or $3p^2$ basic cycles. A p-bit result requires $3p^2/2$ basic cycles but slightly more organisation. Division, using a restoring algorithm, produces a p-bit quotient from a 2p-bit dividend and p-bit divisor in $6p^2$ basic cycles.

In floating point addition and subtraction the timetaken to compare and align operands outweighs the arithmetic by a considerable margin. A scaling operation takes two cycles per bit in vertical mode. Thus, using radix 16 exponent and 24-bit mantissa three normalising shifts are required before and after the add/subtract, and the equivalent of 5 moves to and from workspace, giving 25 basic cycles per bit as opposed to three for fixed point. There is clearly a great advantage in space and time if fixed point arrays of low precision can be used.

The following table summarises the theoretical limits on vertical operations. Practical measures will be given in the next lecture.

TABLE 1: THEORETICAL BOUNDS ON ARITHMETIC SPEEDS
All operands in vertical form

	FIXED POINT	FLOATING POINT
ADD/SUBTRACT	$3p$	$11f + 6e + 4df$
MULTIPLY	$3p^2/2$	$3f^2/2 + 3e + 2f$
DIVIDE	$6p^2$	
MOVE	pc	pc
SCALE	$2p$	$2p$
COUNT	$2p$	

Where:

p is the number of bits in the operand
f is the number of bits in the fraction
e is the number of bits in the exponent
d is the number of denormalising shifts
c is the distance moved in rows + columns

2.2 Data routing

The figures of Table 1 indicate that numerical procedures will be dominated by multiply/divide and floating point add/subtract times: each such operation requires at least 500 DPA cycles, or 100μsec on the assumption of a 200nsec effective execution time. The most efficient use of the array will be achieved in two stages: problem analysis, which seeks to minimise the arithmetic content and external I-O (which will be examined later); and detailed storage mapping aimed at maximum PE utilisation.

Data routing functions are used to move from one store map to another. Although complex at times the intuitive feeling that routing will dominate execution time quite often turns out to be incorrect. In weather forecasting, for example, using spherical mapping of the type described in the first lecture, the routing overhead is estimated to be about 4% of the total execution time. When making comparison with sequential machines it must also be remembered that they too sustain a significant amount of routing overhead in the shape of register load and store, shift and copy instructions. It is important to compare MOPS, ie (millions of) useful arithmetic operations per second, rather than MIPS, ie instructions executed regardless of whether they do anything useful to the outside observer.

Movement within a bit plane and within local storage use distinct mechanisms, so they will be examined separately. For horizontal data, remapping involves movement in the north-south direction and relocation in PE stores, while east-west movement is used for scaling. For vertical data, scaling is effected by relocation in PE stores and remapping involves both east-west and north-south shifts. In converting from horizontal to vertical form (and vice-versa) a rotation procedure is used:

```
REGS[ horiz  vert  temp]
DO (temp ORC  horiz; vert IRC/U temp;vert'N;horiz'1) WHILE VA
```

which is repeated for each p-bit N-vector. Thus mode conversion takes about the same time as multiplication.

Data movement requires one DPA operation for each row or column traversed within the plane. Two extreme examples which are often used as benchmarks are uniform shift or rotation in the plane and arbitrary permu-

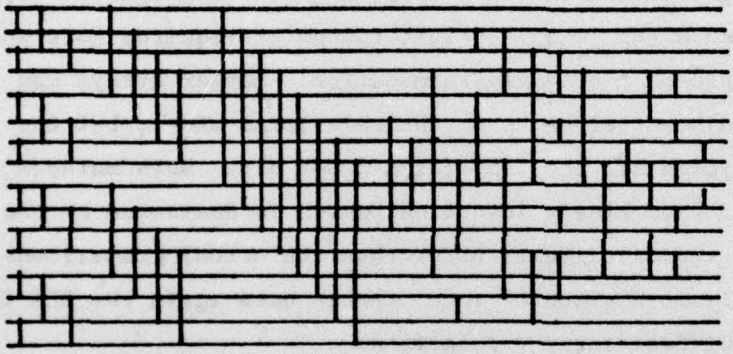
tation of elements, in each case regarding the DPA as a linear array of N^2 cells.

Using cylindrical geometry the average number of operations for a uniform shift (assuming all equally likely) is N/2. In practice, it often appears that not all shifts are equally likely: near-neighbour connections in the plane predominate, with power-of-two shifts occurring quite often. The four near-neighbours are accessed in one cycle, and the eight near-neighbours in 1.5 cycles on average. It is easy to see that in power-of-two shifts the average number of operation is N/n, ie $N/\log_2 N$. In each case the operationmust be repeated p times for p-bit words. If the data array is larger than the DPA it is necessary to use plane geometry, making the edge connections through ACU registers, resulting in three DPA cycles per row or column traversed per plane and considerably greater overhead in control. There is no significant advantage from using data arrays that are smaller than the DPA.

Any permutation of elements can be represented as a sorting problem by attaching a key to each giving its (unique) destination in the final listing. At first sight sorting in unattractive for an array processor because it implies an irregular routing of elements. In a sequential machine the number of comparisons B(t) required to sort t items by binary insertion [6] is $t*\log_2 t - t + 1$ when t is a power of 2, eg B(16)=49, B(32)=129. A 'minimum delay' parallel sort is shown in Figure 5 for t = 16, in which each horizontal line represents an item in the list and each vertical line joins two items to be compared, followed by an exchange if the lower element (in the diagram) is lower in value. The resulting list is in ascending order from top to bottom. It can be seen that in moving from left to right only one or two pairs are being compared and that if 16 processors were available several successive stages could be overlapped. In the example, only 10 distinct stages or delays are used.

The DPA does not have direct routing across several PEs as the minimum delay sort assumes. An alternative is the odd-even exchange, which requires only neighbours to be compared. In the example, the number of stages is 16, which generalises to t for sorting t items. In fact t is an upper limit because the sort is complete whenever a comparison is not followed by an exchange. It is possible that the total number of exchanges

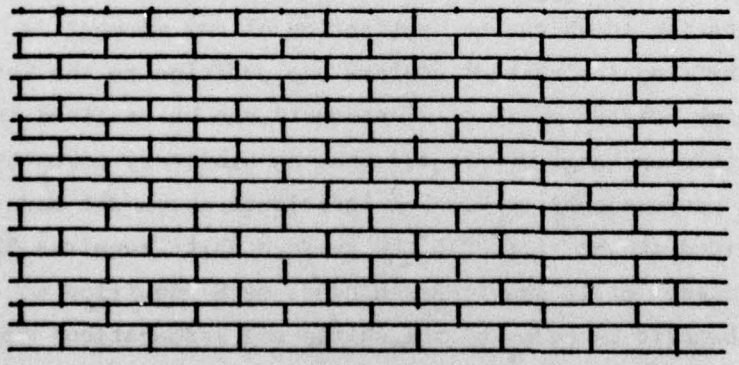
MINIMUM DELAY PARALLEL SORT



N = 16

Delay= 10

ODD-EVEN EXCHANGE SORT



N = 16

Delay = 16

Figure 5: Parallel sorting algorithms

(see [6])

required in practice could be reduced by occasionally sorting in the orthogonal direction, by analogy with Shell sorting: if the N^2 elements are ordered by linear connection in the east-west direction that would imply sorting north-south in order to accelerate progress towards the final positions. I do not know of any practical or theoretical studies of such techniques.

Applying the above results to the DPA, we see that N^2 items of p bits can be sorted in vertical mode in $N^2(6p_k + 3p)$ cycles, where p_k is the precision of the key (comparison takes 3 cycles and exchange 3 cycles per bit). Straightforward insertion, in which a new list is built up in the required order by adding elements one at a time, requires a comparison and move at each step, giving $N^2(4p_k + 2p)$ cycles, though it requires more space and does not offer the prospect of early termination. Larger arrays can be handled by storing adjacent elements in the same local store, but the exchange then takes 4 cycles and the comparison 2 per bit. Both methods are significantly better than binary insertion, which is dominated by the time to move and insert the data items rather than the actual comparisons. The same techniques apply in horizontal mode, though once again the DPA is at a disadvantage without fast carry propagation.

An additional wired interconnection pattern known as a 'shuffle' has been proposed to assist routing operations [6,p237], [7] and [8]. The shuffle effects a permutation in which the destination of any element is defined by cyclically shifting its current address one position to the left. It has been shown that any uniform shift of N^2 elements can be realised in a multiple of $\log_2 N$ shuffle-exchange steps, and that an arbitrary permutation can be achieved in time proportional to N. The individual steps are more complex than those outlined above: for DPA6 the average shift requires 32 moves, ie 32 machine cycles, or 12 shuffle-exchanges, each requiring 4 cycles. The practical benefit in terms of the shifts and permutations most frequently encountered remains an open question. The relevance to Fast Fourier Transform is examined in the next lecture in connection with the DAP.

Evidently there are many applications of DPA to sorting both large and small data sets and as for sequential machines the eventual

choice of algorithmdepends on the characteristics of the data and the way it is used. Before leaving this topic it is as well to recall that the need for sorting must be reviewed at the systems analysis level. It has been stressed in the past because of the limitations of sequential search methods, but given that a DPA can search N^2 items in parallel there may be no point in retaining data sets of less than N^2 items in sorted form: they can be accessed in any desired order. The last comment is particularly relevant where there are multiple keys and the retrieval criterion is a logical or arithmetic function of the keys. A DPA6 would carry out useful searching operations at a rate exceeding 10^{10} bits/second, which is probably one of its most cost-effective application areas.

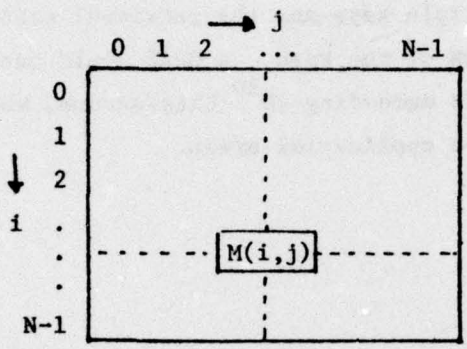
2.3 Matrix operations

Matrices are stored in either horizontal or vertical mode: DPA4 can process a d*16 matrix in horizontal form (or d*32 if the elements are bytes), the local store providing the second dimension d; it can process a 16*16 matrix in vertical form, of any precision up to 128 bits. Larger matrices can be handled by partitioning, but as the resulting algorithm is often expressed in terms of operations on d*16 or 16*16 matrices that is usually best done by creating structures of three or more dimensions the local store providing the third and higher dimensions. A DPA4.1 would contain up to 32 matrices of 16*16 elements in single precision, 32-bit form.

Although the processing rate in either mode is theoretically about the same (with suitable arrangements for carry), vertical mode offers variable precision and indexing flexibility that does not exist for horizontal. One of the disadvantages of single bit PEs in comparison with machines such as Illiac IV is that it is uneconomical to provide local store indexing, but that can be overcome as explained below by using projection operations. In general, the horizontal form is attractive in DPAn for 'vectorial' problems of fairly low precision (up to 2^n) or where frequent word access by the ACU is implied.

For the remainder of this subsection we assume that vertical

data is used, and unless otherwise specified a matrix is taken to have indices running from 0 to N-1 ($=2^n-1$), the coordinate axes being i (north-south) and j (west-east). The precision, or the number of bits in each element, is given by the limit field of the matrix address, plus 1. If the limit is zero there is only one bit and the matrix is said to be 'boolean'.



One method of sorting not touched on in the previous sub-section is by selecting a maximum (or minimum) element, which is eliminated by masking, then the next largest, and so on. The following program selects the largest positive element in a fixed point matrix M masked by a boolean matrix MASK.

```
REGS  [ M  MASK  temp  p ]
/* Find the precision p from the limit of M and set the
activity bits from the mask, eliminating negative values */
p = LIMIT(M); LDA MASK; LDB MASK; AND M'p; EQU -MASK;
STB MASK; temp AOR -MASK; temp % -1; if ZE return; LDA MASK; p-16
/* Now the A registers contain the reduced mask of elements to
be scanned.  In the next loop, the mask is 'anded' with successive
bit planes in M  */
WHILE GE DO (LDB M'p; STB MASK; temp AOR -MASK; temp % -1;
        IF NZ LDA MASK;p-16; RETURN
```

On return, MASK indicates by 1's the position of the maximum elements, if any. The number of DPA cycles is $4p_k+7$ for each selection. Adding the time taken to digitise elements or extract them, the complete sort takes about the same time as those mentioned earlier. In many applications, however,

only the maximum items are of interest.

Projection operations are used to distribute data along row or column lines. The source may be a scalar value in the ACU or a vector selected from store. For example, we can define a procedure ROWP(x,y) that will project x if it is numeric into every element of the matrix y, and if x is a matrix it will extract a vector by column selection (the low order address bits) and project it by row to y. The matrix multiplication Z := X*Y takes the form:

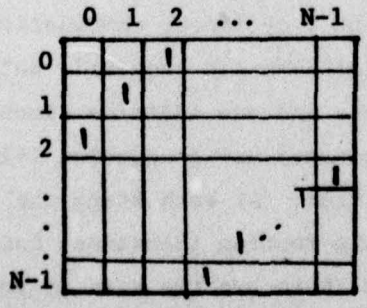
```
            REGS [X  Y  TX  TY  Z  N ]
ROWP (0, Z); DO (ROWP(X, TX);COLP(Y, TY);
                    MULT(TX, TY);ADD(Z,TX); X'1; Y'1; N-1)
                WHILE GE;
```

where COLP is defined similarly to ROWP.

More generally, projection can be based on a vector selected by boolean matrix which specifies by 1's a single-valued boundary to be used in defining a vector. The following statement projects a single bit selected by MASK from the matrix M into TM by row:

```
            REGS [M  MASK temp /* a workplane */ t  TM]
LDB -MASK; OR M; STB/U temp; t AOR temp; TM IR t
```

There are four versions of the code since the ACU allows selection of the control vector by row or column and, independently, projection by row or column.



An example of a selection mask.

Sets of linear equations of the form: $Mx = Y$ may be solved by inverting M, for example by the method of Gauss-Jordan elimination given later (p.43) for the DAP and then premultiplying Y by M^{-1}.

A number of techniques particularly suited to parallel operation have been developed for the solution of tridiagonal sets of equations. Here the matrix M takes the form:

$$\begin{bmatrix} d_1 & f_1 & . & . & . & & & \\ e_2 & d_2 & f_2 & . & . & & & \\ . & e_3 & d_3 & f_3 & . & & & \\ . & . & e_4 & d_4 & f_4 & . & & \\ . & . & . & . & . & . & . & \\ & & & & . & . & . & \\ & & & & & e_{m-1} & d_{m-1} & f_{m-1} \\ & & & & & . & e_m & d_m \end{bmatrix}$$

with zeros off the diagonals. In DPAn it is possible to store 2^{2n} sets of coefficients in vertical form, though the method of reduction ideally requires $m=2^{2n}-1$, so that DPA4 would handle 255 equations, represented by four matrices E, D, F, and Y.

The method of cyclic odd-even reduction eliminates the unknowns of odd index by linearly combining equations, yielding a new tridiagonal system of size $(m-1)/2$. The process is repeated until a single equation is found, which is then solved and the remaining unknowns found by back-substitution.

The numerical algorithm consists of eliminating the coefficient of x_{i-1} in each even numbered equation i by linear combination with equation i-1. Six multiplications and two additions are required, but because a pair of PEs is involved only three multiply and one addition times are required. The coefficient of x_{i+1} is then eliminated using equation i+1, which again requires three multiply and one addition. At each stage the number of elements is halved, therefore the data routing increases, but it can be seen that in the elimination process there are two sets of power-of-two shifts, which are repeated in back-substitution, requiring 4N moves. In

back-substitution an equation of the form:

$$ex_{i-1} + dx_i + fx_{i+1} = y$$

hasto be solved for x_i, requiring two multiplications, two additions and one division, which can again be compressed by using adjacent PEs for multiplication.

To form an idea of the relative magnitudes of arithmetic and routing, we may take the mean time of arithmetic operations to be 200μsec giving 2.4msec per stage, or for N=16, ie for 255 equations, eight stages or 20msec. The number of moves is about 100, which requires under 1msec for 32-bit operands. ie about 5% of the computation time.

The reader will be able to suggest several ways of speeding up the procedure: in later stages of calculation it is possible to increase parallelism by spreading the reduction over more PEs; if several sets of equations are being solved the reduction of one set may be partly overlapped with the back-substitution of the preceding set; and finally the numerical algorithm may converge before completing the reduction,in the sense that the off-diagonal terms are all less than a preset value. The solution of tri-diagonal equations illustrates very clearly the way in which numerical, data manipulation and programming skills can be combined to make efficient use of a DPA.

3 EXPERIMENTAL ARRAYS

We now leave theory to examine three recent examples of arrays of PEs with single-bit data paths: STARAN, CLIP and DAP. Although they share the same engineering technique the position of the array within the system and the organisation of software to support parallelism are quite different in each case. The first two are primarily intended for specific problem areas, namely aircraft tracking (STARAN) and image processing (CLIP), but they have many features of general applicability that I shall use to illustrate alternative design approaches. The reader is referred to the published papers for further information.

3.1 STARAN [10]

The STARAN associative processor can be viewed as a control memory shared by three processors: a PDP-11 host, an array control unit, and an array I-O controller. The function of the host is to handle external communications and to load array programs into the control memory, part of which is fast (150 nsec), the remainder slow (1μsec).

Instructions taken from the control memory by the ACU are broadcast toa linear array of some multiple of 256 elements (in the Rome Air Development Center configuration there are 1024 PEs). Each PE has three single-bit registers and 256 bits of local store. A feature of STARAN is the wide variety of connections that can be made between local stores and PE registers, but first its operation will be described assuming simple local store addressing as for the DPA.

The PE registers are designated X, Y and M. The input (f) to the ALU is one of X, Y, M, a bit from the local store (m) or a data bit (d) broadcast from the ACU. In arithmetic instructions a function $\emptyset$ is applied

to either or both of two pairs of arguments (X,f) and (Y,f), where Ø is one of the sixteen boolean functions of two single-bit arguments. The result of Ø(Y,f) overwrites Y. The result of Ø(X,f) overwrites X either unconditionally or conditioned by the original value of Y, ie if Y=1, X is overwritten, else X is unchanged.

The above instructions can be written in the form:

f Ø g h

where g is the store option on X, ie "X" meaning unconditional write or "X/Y" meaning write conditioned by Y; and h is the store option on Y, ie "Y". Absence of g or h implies that no store takes place. Other array functions are provided to load M and to store Y either conditionally or masked by M, ie m becomes $(Y.M + m.\overline{M})$. The following example of vertical addition is taken from [10].

The problem is to form the one-bit sum B = A + B.

```
/* Initially X=0 and Y is set to the carry-in  */
          1:      A  XOR  X/Y   Y
/* Now X=A.CARRY and Y=A%CARRY */
          2:      B  XOR  X/Y   Y
/* Now X contains the carry, Y contains the sum  */
          3:      Y  XOR  X     B=Y
          4:      X  XOR  X     Y
/* Now X and Y are ready to process the next bit  */
```

Thus serial addition takes 4 cycles, or 800nsec per bit, not counting the ACU overheads.

As already noted, local stores are not directly connected to the PEs. Starting with an array of 256 stores of 256 bits, the stored pattern is skewed through 45° as shown in the diagram for a 4 by 4 array. The advantage gained is that both rows and colomns of the original array can be accessed as words by suitable indexing of each column and some potentially useful 'hybrid' combinations of rows and colomn can be implemented. In DPA terms, if we think of the data as stored in horizontal form it can be processed serially by bit (256 words at a time), serially by byte (32 bytes at one time), and so on.

ORIGINAL			
00	01	02	03
10	11	12	13
20	21	22	23
30	31	32	33

SKEWED			
00	01	02	03
11	12	13	10
22	23	20	21
33	30	31	32

Second column:	11	01	31	21
Third row:	22	23	20	21
Serial-parallel:	00	01	20	21

It will be seen that the words retrieved generally need permuting to appear in the sequence of the original array, and that is done in a separate 'flip' network. The flip network is a permuting device that takes data words read from the local store array or from the words of 256 X, Y or M bits in the PEs and carries out a rotation or reversal on each segment of bits in the input word. A segment is selected by program. It is a power of two (up to 256 bits) in length. The output provides the input f to each PE in the array instructions. Thus, in the example given above, to bring the second column into correspondence with the original form we would take segments of two bits each and reverse them, (11 01) becoming (01 11) etc.

It is difficult to see application for more than a few of the permutations permitted by STARAN. The skewed form of store is clearly helpful in allowing a choice between vertical and horizontal processing without the need to rotate data in the store which, as we noted for the DPA, takes about a multiply time. In a machine with very much faster arithmetic, such as Illiac IV, the ability to skew data assumes greater importance. The power-of-two shifts applied by the flip network are useful in many applications, on the other hand they are of less importance than arithmetic, and it could be argued that faster operation and more local store would be a better investment for general purpose array work.

An extra facility that is valuable in search procedures is the detection in any array of 256 PEs of the index of the first non-zero Y bit, if any. It is in that sense that the array can be labelled 'associative'. Without the additional hardware, eight mask and compare operations would be needed to develop the digits of the index.

3.2 CLIP [11]

The processing of digitised pictures introduces a class of problems not considered so far. If an image is represented by a grid of black and white dots then the recognition of the boundary of a two-dimensional object involves much more subtle neighbour interaction than has so far been discussed. Picture processing machines can be thought of as distributed processor arrays with a well developed means of propagating signals across the array. For example, to detect a closed boundary marked by 1's we could 'flood' the array with 1's input at the edges and allow the signal to spread until a boundary is reached, then stop. The boundary points can then be marked by 'anding' the original image with the occupied cells.

The local PE connections assumed are usually 6 or 8 neighbour, with programmed selection of the rule of signal propagation. In a rectangular DPA any interconnection pattern can be programmed with the help of explicit move instructions, whereas in a picture processor the signal is allowed to 'ripple' through the PEs (by analogy with carry propagation in horizontal mode) in a single instruction of variable duration. Each picture element (pixel) is mapped into the local store of one of the PEs: DPA4 would represent in one bit plane a 16*16 black and white image, or d*256 if the pattern is stored vertically. Larger pictures, grey code or colour images would naturally require more storage. In addition, working storage is needed in each PE to contain derived patterns representing boundaries, internal regions, etc.

In a typical image transformation a single bit in each element is designated the 'output'. It is formed according to the current state of the element, ie PE register values, and inputs received from selected neighbours. For example, the transformation rule written as:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & s_i & 0 \\ 0 & 0 & 0 \end{bmatrix} \longrightarrow 1, s_j$$

can be read 'if in state s_i and the input from the NW neighbour is 1 and

all others are zero, output 1 and go to state s_j'. The transformation is applied in parallel to all picture elements and repeated until there is no change in the output pattern for the entire image (one way to ensure termination is to allow only 0→1 changes in the output). For example, if $s_i=s_j$ the above rule would propagate a diagonal line of 1's from any 1 input on the north or west edge, until a cell not in state s_i, or in a 3 by 3 region containing a 1 apart from the NW corner is encountered.

The CLIP array described in [11] consists of 16*12 PEs specialised to the type of transformation just described. Each PE has two single bit working registers A and B, an output N, and 16 bits of local storage D. There are three types of array instruction:

LOAD: Initialise A and B, using the local store or zero as input. The geometric pattern (square or hexagonal)is also specified.

PROCESS: Apply a transformation rule using the inputs N until there is no change in N throughout the array. Any of the neighbour connections can be selected and summed, then compared with a threshold value t. The PE input T is set to 1 if the sum exceeds the threshold, else zero. The value of N is $\emptyset(B \vee T, A)$ where $\emptyset$ is one of the 16 boolean functions of two variables. The PROCESS instruction also selects the edge inputs.

STORE: A boolean function $\emptyset'(B \vee T, A)$ is evaluated and the result written to a local store plane D_i or combined with the current value of D_i by 'and' or 'or' operation.

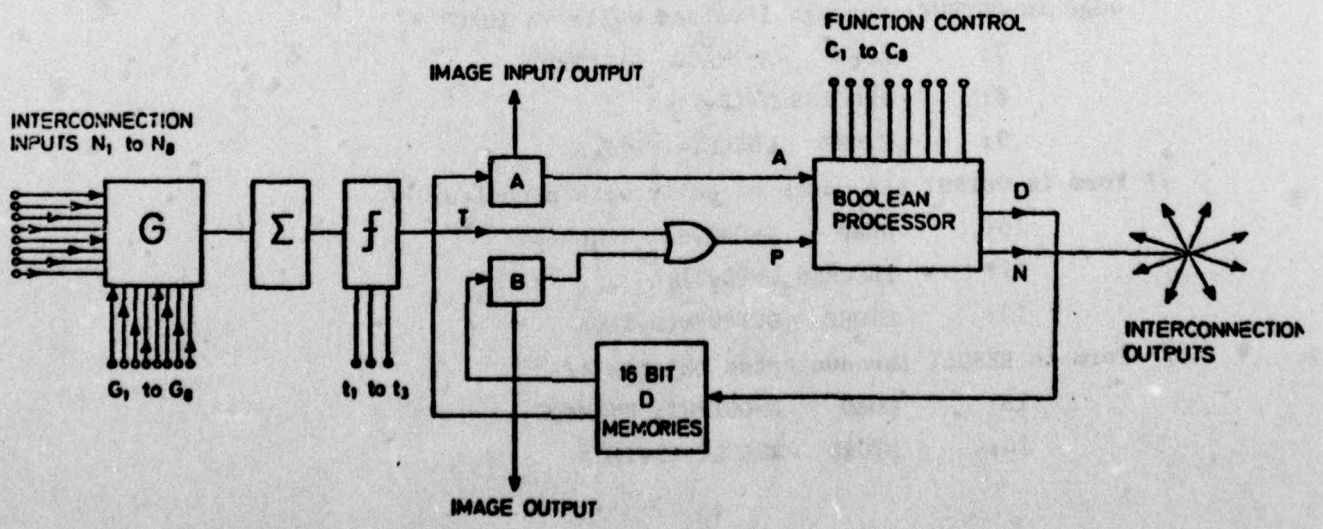


Figure 6: PE schematic for CLIP-3

In addition to the above instructions the ACU, which is controlled by a separate 256*24-bit memory, can execute subroutine calls using a 16 word link stack, branch or branch conditional on the AND of N outputs taken over all 192 picture elements. Provision is also made to display the A and B planes on a CRT so that the effect of different algorithms can be observed experimentally.

The folowing example is taken from [11]. Given an image containing biological cell patterns, it is required to select the outlines of all cells containing nuclei. Hexagonal connection is assumed, with all six inputs active and threshold zero. In the following symbolic program IMAGE, OUTPUT, etc, refer to bit planes in the local store D_i and the notation is chosen to give the flavour of the calculation rather than detailed instruction formats. Figure 7 shows the working results obtained after each STORE instruction.

```
/* From in OUTPUT the outer edges of objects in IMAGE  */
          1:      LOAD     A=IMAGE; B=0
          2:      PROCESS  N=(B∨T)∧Ā; Edge input = 1
          3:      STORE    OUTPUT=(B∨T)∧A
/* Form in GROUND the background surrounding IMAGE objects */
          4:      LOAD     A=IMAGE; B=0
          5:      PROCESS  N=(B∨T)∧Ā; Edge input = 1
          6:      STORE    GROUND=(B∨T)∧Ā
/* Form in NUCLEI the cell nuclei.  Propagation starts from the outer
   edge in OUTPUT, through 1-valued cells in IMAGE */
          7:      LOAD     A=IMAGE; B=OUTPUT
          8:      PROCESS  N=(B∨T)∧A
          9:      STORE    NUCLEI=(B∨T)‾∧A
/* Form in OUTPUT the masks of cells with a nucleus */
         10:      LOAD     A=GROUND; B=NUCLEI
         11:      PROCESS  N=(B∨T)∧Ā
         12:      STORE    OUTPUT=(B∨T)∧Ā
/* Form in RESULT the nucleated objects */
         13:      LOAD     A=OUTPUT; B=IMAGE
         14:      STORE    RESULT=(B∨T)∧A
```

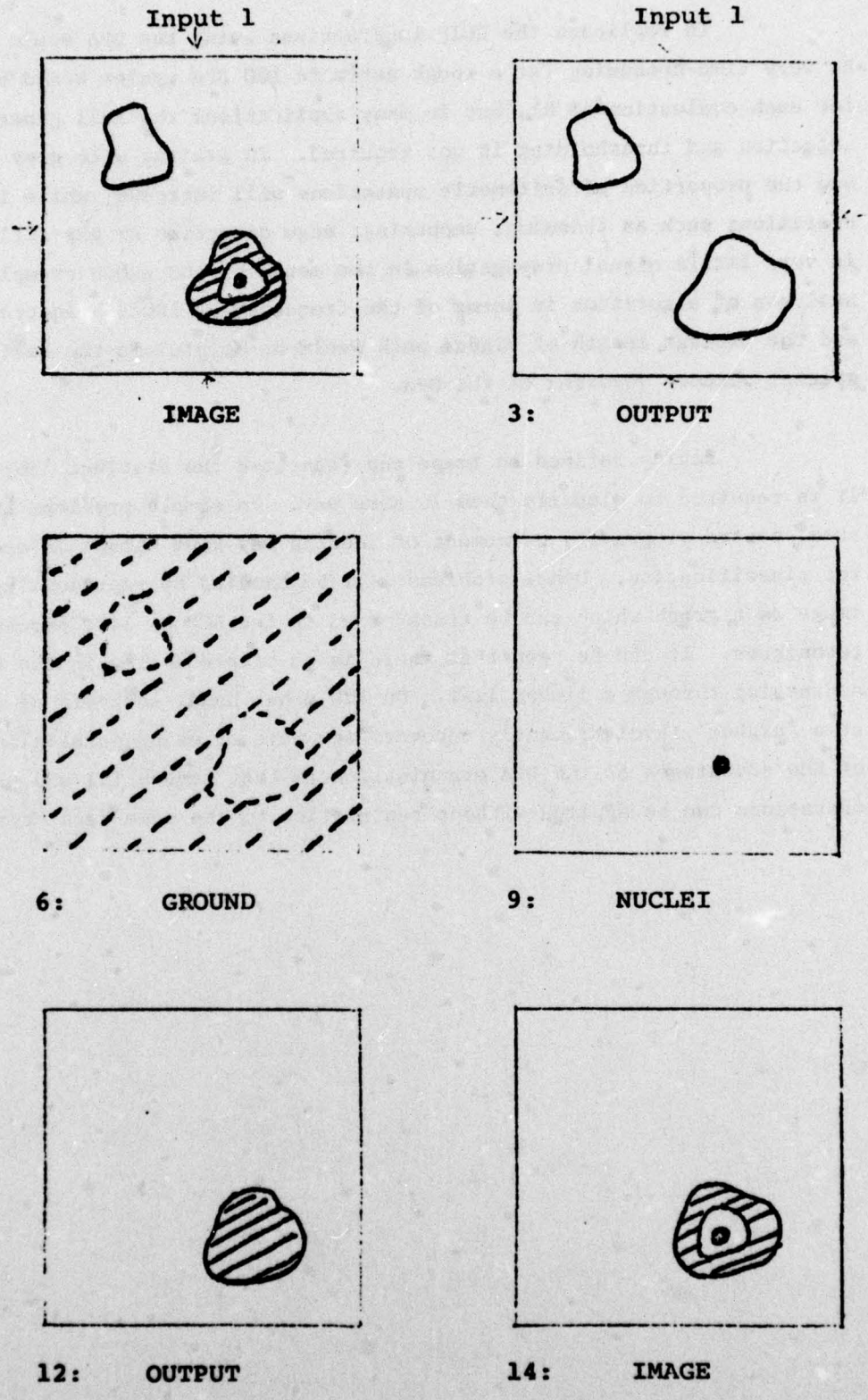


Figure 7: An Example of CLIP Processing

To replicate the CLIP instructions using the DPA would clearly be very time-consuming (at a rough estimate 100 DPA cycles would be needed for each evaluation of N), but in many applications the full generality of selection and thresholding is not required. In dealing with grey scale or hue the proportion of arithmetic operations will increase, while in many operations such as thinning, smoothing, edge detection or gap filling there is very little signal propagation in the sense of the above example. An analysis of algorithms in terms of the frequency of PROCESS instructions and the average length of signal path would be helpful in the design of special purpose versions of the DPA.

Having refined an image and separated the distinct 'objects' it is required to classify them in some way. In simple problems the area, centre of gravity or moment of inertia may give enough information for classification. Other problems will be handled by representing the image as a graph which can be transformed in the ACU by list processing techniques. It can be seen that there is no simple analog in the DPA to addressing through a linked list. On the other hand, analysis of problems at a 'higher' level frquently uncovers new ways of using parallelism. One of the advantages of the DPA organisation is that sequential and parallel operations can be applied without restriction to the same data sets.

3.3 DAP [12] [13]

The reader will recognise the ICL Distributed Array Processor as the experimental model from which I have extracted the principles of the DPA. It differs from DPA5.2 in details of PE and ACU design.

In the PE (Figure 8) the four near-neighbour shifts are incorporated into the arithmetic functions, so that it is possible to take an operand from any of five PEs. The destination is local and controlled by the activity register (A) as for the DPA. The A register can be used in its own right for general logical operations, in particular for combining boolean activity matrices. Data movement is carried out between PE registers rather than stores. Provision is made for ripple carry propagation in the *east-west* direction.

The resulting srithmetic speeds are shown in Table 2. with the contribution of data and instruction accesses (in DAP each instruction occupies 32 bits). It can be seen that despite using horizontal carry the vertical mode remains more effective when the required level of parallelism can be achieved: that is the consequence of the carry propagation time and the higher overhead on normalising shifts in horizontal mode. The effect of using specialised procedures for square and square root is apparent from the times given.

The relatively low instruction access counts shown in Table 2 are the result of buffering in the ACU, which is explicitly controlled by program, ie by a 'DO...REPEAT' construction which marks the beginning and end of each loop. Within the loop, instructions are not only buffered but provision is made to increment or decrement address fields on each iteration. The buffering mechanism reduces the instruction fetch overhead to about 10% on elementary arithmetic and logic and 25% - 40% on multiply/divide and floating point. Outside the loops, instruction overhead is at least 100% of data access. Where there is high arithmetic content, most of the computation is within loops, eg taking matrix inversion (29msec) and subracting the time for finding the pivot, add, multiply and divide leaves only about

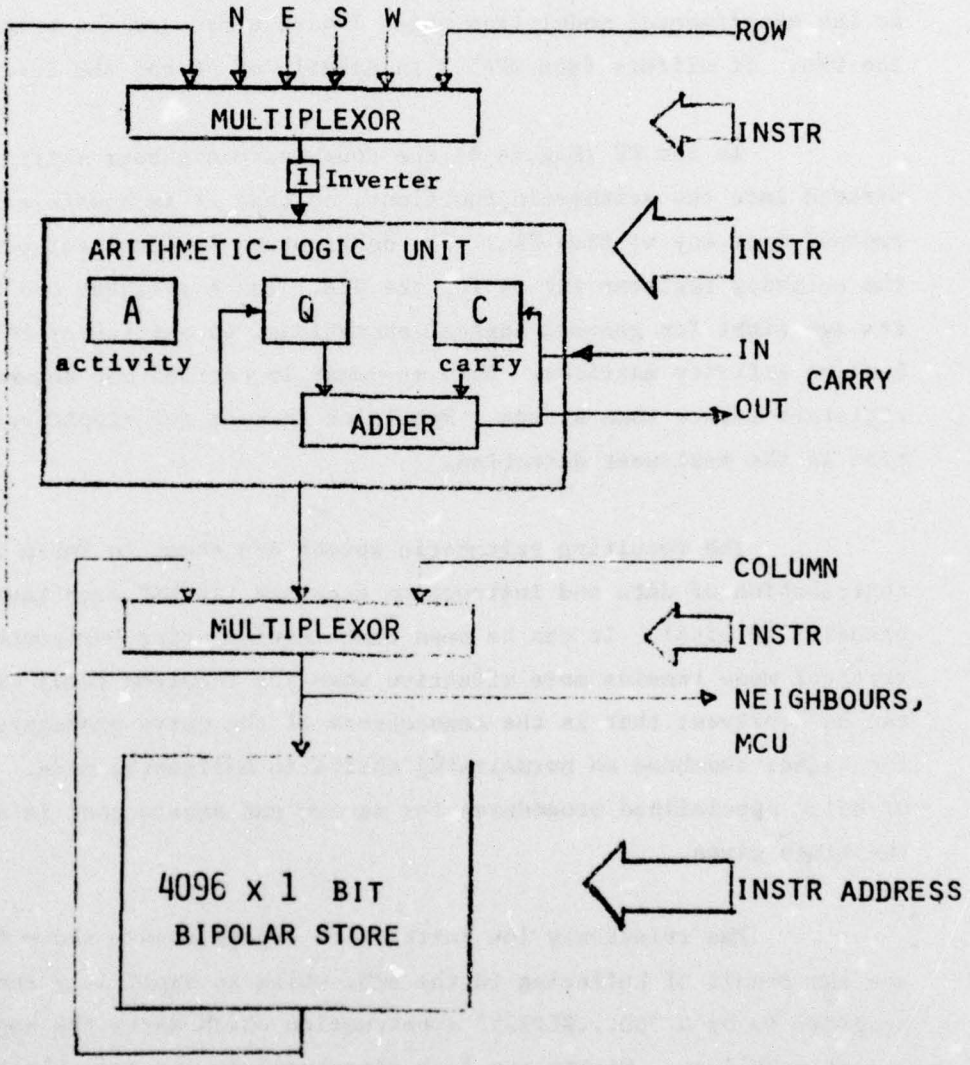


Figure 8: Processing element schematic for DAP

TABLE 2: MEASURED EXECUTION TIMES FOR DAP 32*32 PEs

All times in μsecs		MATRIX (1024) TOTAL	INSTR	EFF	VECTOR (32) TOTAL	INSTR	EFF	SCALAR (1) TOTAL	INSTR	EFF
32-bit FIXED POINT										
R := P + Q		23	3	.022	4		.125	4		4.
R := P		14	2	.013	1		.031			
R := MAX(P,Q)		34	2	.033						
32-bit FLOATING POINT										
T := X + Y		148	26	.145	54	22	1.69	27	12	27.
T := X * Y		305	110	.298	50	10	1.56	34	14	34.
T := X / Y		390	120	.381	100	20	3.13			
T := X ** 2		155	60	.152	40	10	1.25			
T := SQRT(X)		215	70	.210						
SCALAR-MATRIX										
X := S*Y	min	40	10	.039						
	max	150	50	.146						
S := SUM(X)		165	10	.161						
S := MAX(X)		46	2	.045						

Note: 'EFF' is the effective time for single operands, ie TOTAL/parallel data streams.

MATRIX OPERATIONS

MULTIPLY(X,Y)	16msec
INVERT(X)	29msec
FFT(X)	14msec

[All 1024 element single precision floating point arrays]

100 organisation instructions on each iteration. Instruction fetch overhead is reduced in larger arrays and could be eliminated by using separate control storage: the engineering trade-offs are essentially the same as for microcode.

The Fast Fourier Transform algorithm is often used to justify additional routing capability. In DAP it is applied to an array of 1024 complex values or to a two-dimensional 32*32 array, in each case in vertical mode. For 2^{2n} variables, 2n parallel computing steps are required. The routing pattern for n=4 is shown in Figure 9. In general, the first step can be carried out in one cyclic shift, the remainder need two shifts each. Using orthogonal connections, the number of complex moves is $3*2^n+2^{n-1}-4$. After completing the transformation a second series of n shifts is required to return the elements to their original positions. The total number of moves is again proportional to 2^n. A DAP program, taking advantage of the

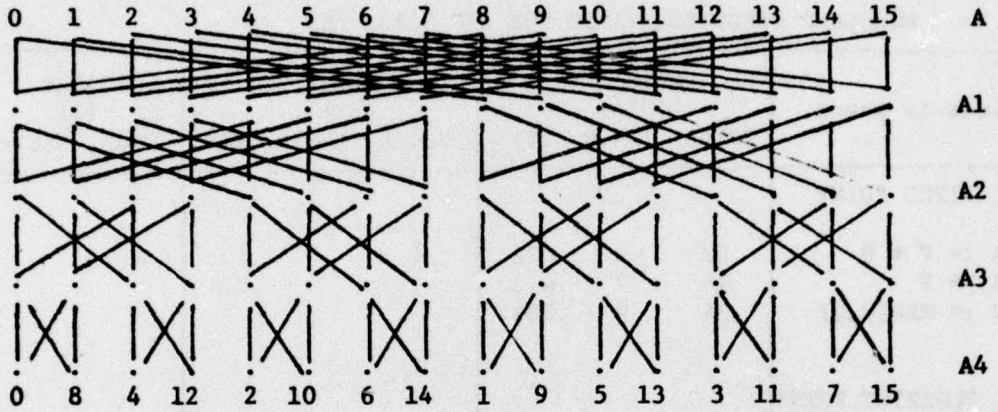


Figure 9: Data routing in the FFT for n=2 (2^{2n} variables)

simple form of multipiers in the early stages of calculation, but deriving successive mutipliers by a recurrence relation, has the following contributing factors:

	Count	Time	Total
Multiplication (32 bit fl.pt.)	32	305 μsec	9.76 msec
Addition (32 bit fl.pt)	38	148	5.62
Assignment	90	15	1.35
Routing	216	7	1.51
Compute multipliers	16	500	8.00
		Subtotal	26.24 msec
Reshuffle	286	7	2.18
		Total	28.4 msec

(In the first step A1 is calculated: A1(0) is the sum of A(0) and the product of a complex multiplier (in this case 1) with A(8); in the next step A2(0) is the sum of A1(0) and a multiple of A1(4), and so on. After four steps the original A(15) has been routed to contribute to A4(0) and A(0) to A4(15)).

From the above figures it can be seen that routing is not a major factor for vectors of size 1024. For arrays of 4096 PEs the routing overhead doubles whereas the computation increases by two steps, so we must be cautious in drawing general conclusions. The FFT time given in Table 2 is the result of using coding tricks to reduce the arithmetic content, with the result that routing occupies the DAP for about 25% of the FFT procedure.

The main difference between the DAP and DPA is the role of the array in system: the ACU-DPA could be regarded as a stand-alone processor-memory pair or a node in a distributed system, whereas the DAP is seen as a substitute for a main store module in a conventional centralised system. The control unit of the DAP is concerned only with issuing array instructions and serving requests received over the main store data and address lines (Figure 2, page 7). In DPA terms the 'main store data and address lines' could be replaced by 'interprocessor bus'.

The DAP is therefore a componenet of a larger system, in which the host processor takes responsibility for store management and DAP scheduling and provides all necessary support functions. Tasks are issued to the DAP in the form of 'DAP segments' containing all necessary programs and data. The DAP operates in parallel with the host, serving external requests by interrupt processing and able to interrupt the host on task completion. It is prevented from overwriting store outside the current segment by setting base and limit registers. Although scalar operations can be carried out in the DAP (Table 2) the effect of such an organisation is to concentrate parallel phases of computation into 'DAP subroutines' and to leave the rest to the host. Because of the overhead in forming a DAP segment and scheduling its use there is a lower limit of complexity in what is worth considering as a DAP subroutine, eg we would not use the DAP for looking up a single word in a dictionary, which would be natural for the DPA. The difficulty might be overcome by 'batching' requests for elementary operations, but that tends to complicate software design.

The design of DAP subroutines has followed much the same lines as Illiac IV, for much the same reasons: a macroassembler for basic software and a Fortran-based higher level language. Purists may think that a retrograde step, but at the present stage of development it is important to have precise control of store allocation and alignment as well as processor synchronisation, protection and error management. At some future date, when bit planes are more plentiful, we can afford to be more adventurous.

Figure 10 is an example of a DAP-Fortran subroutine taken from [13] with explanation of the conventions used in indexing.

```
01  C   DECLARATIONS
02      SUBROUTINE INVP(A)
03      REAL A(,), B(,)
04      LOGICAL PROW(,), PCOL(,), PMASK(,), PIVOT(,),MASK(,), PIVOTS(,)
05      INTEGER RN()
06  C   NOTE THAT THE ARRAY DIMENSIONS ARE IMPLICITLY GIVEN BY THE
07  C   SIZE OF THE DAP.  A AND B ARE REAL SINGLE PRECISION MATRICES
08  C   IN VERTICAL FORM, RN IS A VECTOR AND PROW, PCOL ETC ARE
09  C   BOOLEAN MATRICES.
10
11  C   INITIALISE MASK TO CONTROL SEARCH FOR PIVOT ELEMENT
12  C   AND PIVOTS TO MARK THOSE PIVOTS ALREADY USED
13      MASK = .TRUE.
14      PIVOTS = .FALSE.
15
16  C   MAIN ITERATION
17  C   FRST, MAXL AND ABS ARE INTRINSIC MATRIX FUNCTIONS
18  C   EG MAXL FINDS THE MAXIMUM ELEMENT(S) IN AN ARRAY UNDER A
19  C   SPECIFIED MASK
20      DO 1 K = 1,DAPSIZE
21      PIVOT = FRST(MAXL(ABS(A), MASK))
22      S = A(PIVOT)
23      PIVOTS = PIVOT .OR. PIVOT
24      PROW = BYROW(ORR(PIVOT))
25      PCOL = BYCOL(ORC(PIVOT))
26      PMASK = .NOT.(PROW .OR. PCOL)
27  C   BYROW,BYCOL ARE PROJECTION FUNCTIONS
28  C   ORR, ORC FORM BOOLEAN VECTORS BY "OR" OF ROW, COLUMN
29      A(PIVOT) = 1.0
30      A = MERGE(A, 0.0, PMASK) - A(,*PCOL)*BYCOL(A(PROW)/S)
31      PROW = -A
32  1   MASK = MASK.AND. PMASK
33  C   NOTE THE USE OF MATRIX INDEX IN 29 AND PROJECTIONS IN 30
34
35  C   THE FINAL STATEMENTS RESHUFFLE ROWS AND COLUMNS
36      RN = ROWN(PIVOTS)
37      DO 2, K = 1,DAPSIZE
38  2   B(K,) = A(RN(K),)
39      DO 3, K = 1,DAPSIZE
40  3   A(,RN(K)) = B(,K)
41      RETURN
42      END
43
```

Figure 10: DAP-FORTRAN subroutine for matrix inversion

In each iteration the largest pivot element A(p,q) is found and used to compute the new values:

```
                A(i,j)  :=  A(i,j) - A(i,q) * A(p,j) / A(p,q)
row p:          A(p,j)  :=  A(p,j) / A(p,q)
col q:          A(i,q)  := -A(i,q) / A(p,q)
pivot:          A(p,q)  :=  1 / A(p,q)
```

4 SYSTEM DESIGN

Table 3 gives some idea of the DAP performance relative to other 'high speed' machines. All the times are experimentally measured with the exception of DAP-FORTRAN, which is estimated by doubling the control overhead (12msec) of the assembler. The corresponding time for a 64*64 DAP would be about 30msec. Using double precision floating point we expect multiplication to increase as the square of the length of fraction, and addition to be linear, ie from Table 2:

Multiply: $(\frac{56}{24})^2 * 195 + (110/2) = 1117\ \mu sec$

Add: $\frac{56}{24} * 122 + (26/2) = 298\ \mu sec$

Hence matrix multiply increases to about 90msec.

TABLE 3: RELATIVE PERFORMANCE MEASURES
MATRIX MULTIPLY 64*64 arrays — All times in msec

Machine	Precision	Assembler	'Fortran'
ILLIAC IV	64 bits	38	60
CDC 7600	60 bits	77	168
IBM 360/195	64 bits	70	110
ICL DAP (32*32 PEs)	32 bits	128	(est)140

Many factors have to be taken into account in estimating relative performance over complete applications, but although it will be argued that I have chosen the most favourable possible comparison for array processors it is certainly not the case that comparisons get progressively worse from the point of view of the DAP: in many major applications a high degree of parallelism can be extracted by careful program analysis. The cost of doing so is no more than a sequential machine would require, once conventions have been established to facilitate thinking in array terms. However, the most vital statistic that might have been added to Table 3 is that the DAP

uses less than 100 000 TTL gates for the entire ACU and PE logic, whereas all the others use upwards of 1 000 000 fast ECL gates.

In the first lecture I said that we were looking for a T% improvement in system throughput for an investment of substantially less than T%. Now system throughput is largely determined by the rate at which tasks are executed and the time remaining after the operating system has completed its business of compiling, loading, scheduling, table maintenance, archiving, spooling, etc. In this lecture I shall examine ways in which the presence of a DPA might affect this negative contribution to throughput by the operating system. Many applications such as searching, indexing and encryption come to mind. However, it might be said that in all but pathological cases the net system overhead is only a few tens of percent of real time and that imposes a limit on potential improvements. My belief is that in system design, as in other application areas, the preferred approach is to start with a restatement of objectives that allows the array to influence subsequent problem analysis. The subsections that follow illustrate how resource management, program context, and higher level connectivity are influenced. But let us first obtain an estimate for the other side of the inequality: the investment in extra hardware implied by a DPA.

The 32*32 DAP is made from standard TTL dual-in-line integrated circuits (DILICs) mounted on boards with about 100 package positions. In the initial design there are 16 PEs to a board, averaging 3.6 DILICS plus two 1Kbit store DILICs each. Hence, to the extent that hardware cost is determined by package count the PE logic represents more than half the board space, compared with purely passive store (the same boards used as control memory provide 64Kbits of storage).

To improve on that picture we must follow up the original intention of using custom-built LSI for the PEs. For the purpose of making comparisons an 'exchange rate' has to be fixed between the PE logic and storage bits, which I shall take to be 128bits(bipolar) = 512bits(MOS) = one PE, based on approximately 50 gates/PE in the DPA design. We assume that at any point in time the PE array will be subject to the same level of integration as the stores.

The limiting factors are the complexity of circuit and the number of edge connections required. For example, a 4*4 array of PE logic is equivalent to 2Kbits of bipolar storage so it is well within the range of current LSI devices. For such a package 16 bidirectional data connections are needed to give row, column and neighbour I-O under control of 3 function bits (data lines are also used for control signals). The addition of parity (4bits), voltage (2),clock(1) and store write-enables (16) brings the pin count up to 42. An alternative is to integrate part of the local store with the PE array: 8Kbits of fast storage in addition to the PEs requires more advanced technology, but the 16 write-enable outputs are replaced by 9 address bits(512bits/PE), allowing greater freedom in pin allocation, ie using more function inputs and relying less on decoding in the device.

Whether or not the local memory is integrated with the PEs it is likely that in future designs the DPA will be enlarged by the addition slow (MOS) storage to the array. A possible configuration would be DPA4.1 (with 32Kbytes of fast store) and an additional 16Kbits of slow store for each PE. The fast memory is now a slave or cache for the 'main store' of $\frac{1}{2}$Mbyte: a bit plane (32 bytes) can be accessed in one memory reference (say 400nsec), the theoretical transfer rate being about 75 Mbytes/sec. A number of architectural questions need to be answered before one can pick the 'best' configuration, but a useful comparison can be made between the enhanced DPA4.1 and a conventional system with $\frac{1}{2}$Mbyte of main memory and 32 Kbytes of fast stores of one sort or another: the PE logic is equivalent to adding another 4 Kbytes of fast store*, and it is that figure together with the LSI development cost, seen as a percentage of the total system, which determines the T% investment I assumed initially.

The following subsections continue to use DPA4 as a model for discussion, but it should be clear where proportionate increases in cost or performance can be expected from larger systems.

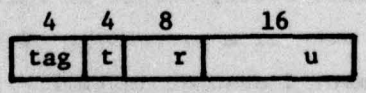
---

* in terms of logic or power; it is more (about 16kbytes) in terms of board space, so the true figure is perhaps in the region of 10kbytes.

4.1 Index management

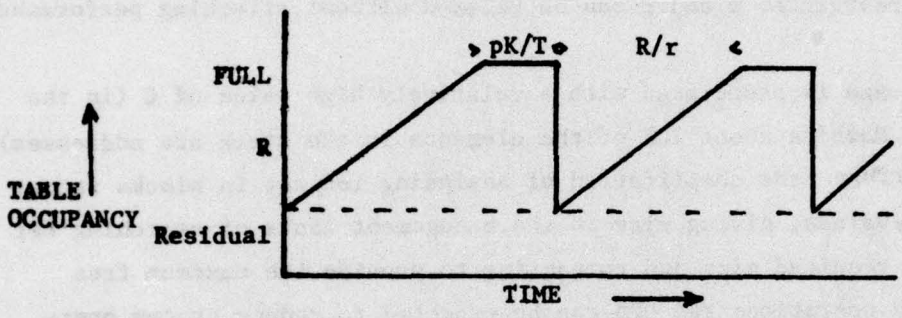
This subsection is concerned with the management of 'objects of computation', in particular with the problem the DPA creates for itself by having two levels of program storage. The topic is important in the design of operating systems because the most effective way of keeping control of complex software structures is to express their procedures in terms of abstract objects (such as files, processes, stacks) whose integrity is preserved by protection mechanisms. In static environments much of the protection - allocation, type checking, etc - can be done at compile or load time, but in-line control is necessary for changing data structures. Of the two methods of control used in practice, ie capability and access control list, the former provide the most precise and efficient treatment. Tagged registers, such as those of the ACU (page 12) are the most flexible way of handling capabilities.

The general system objective is as follows: given a set $[t_i]$ of object types we need to create instances of objects and to assign attributes to them; to grant and revoke access on a selective basis; and to remove objects from the program space when they are no longer required. A capability identifies an object u of type t and rights r by encoding it as a tagged element, eg in ACU4:



where t and r are taken care of by a combination of hard and soft interpretation. Our main concern is with the choice of u, which is either a store index (ie a location number) or an index in a 'master object table' M_i for type t_i. The difficulty is that the index u cannot be re-used until all capabilities containing u have been annulled, and in that sense the management of abstract objects can be viewed as the management of a small number of 'index spaces' where the indices are spread over some fraction of the total program space.

If we plot the occupancy of a master object table we see that it increases with time (at an average rate r indices/second) until the table is full, at which point recovery procedures are invoked to create a new 'free index list'. If R is the number of indices recovered then recovery takes place after R/r seconds.



The recovery process involves scanning all capability-bearing regions of store. The criterion for recovering an index may be that the reference count is zero, or that an explicit 'deletion' operation has been applied. The normal procedure in either case is to take each capability of class t_i and compare it with table entry $M_i(u)$, marking the table or the capability as appropriate. If the total program store is K bytes and the proportion that has to be scanned is p then the recovery time is linearly related to pK/T and pKC, where T is the rate of scan and C is the probability of finding a capability of the given type. The time wasted in index management is expressed as a proportion of computing time by the ratio:

$$W = \frac{pKr}{R}\left(\frac{1}{T} + C\right)$$

The normal methods used to reduce W include:

(a) increasing R, eg using virtual indices in the case of store access;

(b) restricting p by limiting the number and size of capability-bearing segments;

(c) partitioning the program space according to process number, so that smaller regions are scanned (and the cost can be transferred to the process);

(d) reducing r by requiring logically distinct objects to be mapped into the same object space (so defeating one of the aims of abstraction).

The effect of DPAn is to increase the nominal rate of scan, T, by a factor

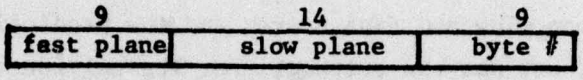
2^n, so that the first term of W is correspondingly reduced. It appears that the individual comparisons with M_i have to be done sequentially, so the second term is unchanged and the benefit of the DPA will be most marked for object types of fairly low population. If W is already small this will be seen not as an increase in throughput but as a change of program style: the measures that restricted p and r can be relaxed without affecting performance.

Storage is associated with a relatively high value of C (in the Basic Language Machine about 20% of the elements in the stack are addresses). Storage also brings the complication of assigning indices in blocks rather than as single values, giving rise to the management tasks of searching for a block of the required size and compacting to provide the maximum free block. In both operations the DPA can be expected to reduce system overheads by direct application of parallel search and relocation procedures. In that sense the presence of a DPA, which we equated earlier with the addition of a small amount of fast store, may in fact reduce overall store requirements: store can be allocated in smaller units, the resulting structures can be managed effectively in less space, and the need for remapping from virtual to real indices is practically eliminated.

Modifications to the addressing mechanism to allow access to slow storage have not been studied in detail. Two possibilities can be suggested. In ACU4, with 16 bit location fields, there is not range enough to cover the slow store, therefore a new type of address is introduced, resolving to the bit plane boundary. The DPA arithmetic functions apply to 'slow' addresses, but the only routing operations are unselective store (STA/U and STB/U). The slow store provides a 'segment space' holding the data and procedures of all programs, which will be mapped into fast store under control of low level interpretive code. It follows that the fast store must be large enough to contain all the 'working segments' of all active processes. The program store K is less than 32Kbytes in DPA4.1, and the proportion of address-bearing segments p is not usually more than 10%. The time of scan is therefore very short in absolute terms. For all other indices the entire program space of up to $\frac{1}{2}$Mbyte is available to K.

An alternative strategy is to use a longer address word, eg a 32 bit location number in ACU6, that can cover the entire slow storage range.

Suppose we have an extended DPA6 with 16Kbit slow stores and 512bit integrated fast store. The loading rule is to allocate a bit plane to fast memory only when its address is formed in an ACU register. There is only one type of address, which contains the location in slow memory, but when the corresponding plane is paged into fast memory its plane number is added to the address:



Consequently, all memory references by the ACU are to the fast store. The address is not updated until the plane number changes as the result of modification. A 512 entry associative store is needed to translate from slow to fast plane numbers: the DPA can perform that function, though in a high performance system specialised stores are probably justified. One purging strategy is to write back to slow store all planes that are not write-protected, and to scan addresses to clear the fast plane numbers and force reloading. Here K is 8Mbytes, and assuming 10% address-bearing up to 1600 planes have to be scanned.

Figure 11 illustrates the two methods of addressing. It is probable that the second is as effective as the first although it uses less fast store.

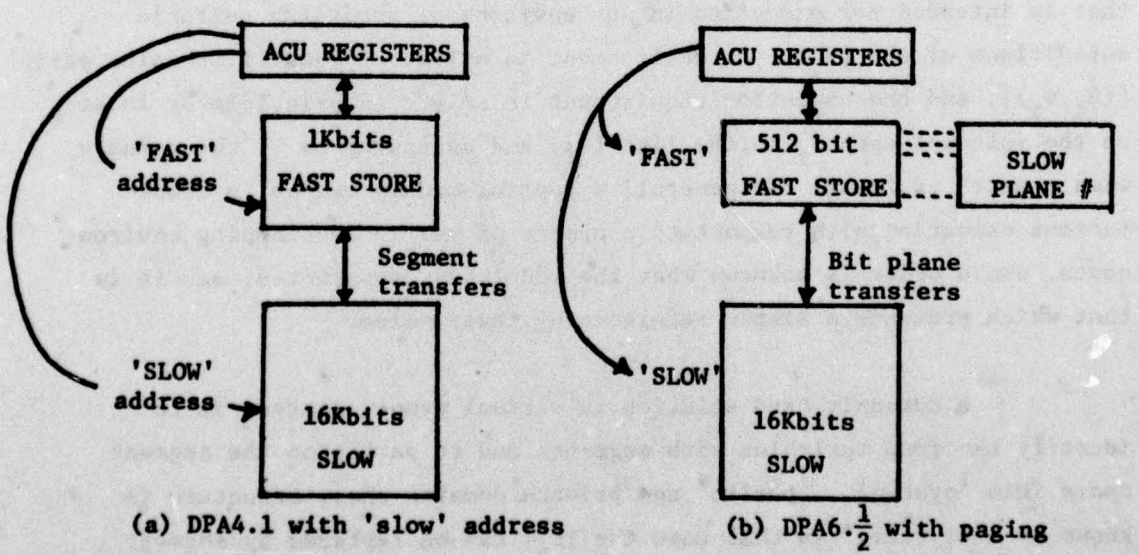


Figure 11: Two methods of addressing a large program space

4.2 Program context

The methods just outlined provide a rapid means of adjusting the content of fast memory to meet program requirements. They use the high bandwidth between slow and fast stores and the predictive property of tagged addresses, but not the arithmetic or logical functions of the PEs. The effect of the DPA in system is thus comparable with other slave memories, given the page size fixed by the bit plane: advantage is taken of locality of reference to data, data descriptors and instructions. If the instruction takes the form of a language-oriented token string there is no need to rediscover the locality because it is already explicit. The DPA is well adapted to interpretiveprogramming techniques. In particular, it solves the system problem of providing fast access to (micro) instructions without having a dedicated control memory.

We now consider using the associative function of the DPA in conjunction with program design. The most suitable areas of application are the interfaces between control modules and between procedures. A control module is a segment of instructions, data and free variables $[F_i]$ that is intended for execution in any environment providing suitable definitions of the $[F_i]$. An environment is a list of identifier-value pairs $[(G_i,v_i)]$, and the execution requirement is solved in principle by looking up the value of each F_i in the list $[G_i]$ and assigning to F_i the value v_i when a match is found. In general, a control module can be in simultaneous execution with respect to a number of partly overlapping environments, whose order is unknown when the module is constructed, and it is that which prevents a simple reference by index value.

A commonly used solution in virtual memory systems is to identify the free variables with segments and to partition the segment space into 'system' , 'public' and 'private' domains whose structure is known by load time. In that case the $[F_i]$ can be replaced by segment indices. Another approach is to carry out the association of $[F_i]$ with the $[G_i]$ in each environment of interest and to store the results in tables that are referenced indirectly via process-dependent addresses.

The disadvantage to such techniques is that they impose unnecessary structure on programs.

The DPA allows a return to the direct and elegant solution: the identifiers F_i are retained in the control module (possibly in coded form) and associated in parallel with the set $[G_i]$. An additional operation that is important in some protection regimes is to check the name of the calling module against an access control list for the callee: that can also be done in parallel.

At the procedure interface similar options apply, except that the identifier-value list is formed when executing the calling sequence. The effect would be to allow parameters to be called 'by identifier'. Although such a facility is attractive in some applications it is unlikely to replace the conventional method of indexing relative to a parameter pointer. A more general approach might be to introduce a class of abstract data types of the form 'identifier-value list', which could be maintained efficiently by the DPA.

4.3 High level connections

The I-O subsystem is a major part of Illiac IV, STARAN and CLIP, yet it has not featured in DAP or the general discussion of DPA's. The primary I-O channel for the DPA is the main store highway, the maximum transfer rate via the ACU being 5Mwords/second, the sustained rate naturally depending on the bus capacity. Assuming 1Mword/sec and using DPA6, a matrix of 32 bit planes can be input in 2msec or about 10 floating point operation times. To a first approximation we can derive conditions on any application for 'balanced' computation and data flow. As the capacity of local storage devices increases so does the range of problems that can be contained wholly in the array; for example, the addition of a 64Kbit serial CCD store to each PE in DPA6 would extend the internal storage to 40Mbyte, which gives the PEs quite a lot to work on.

Higher I-O rates are required in the context of increasing the processing power by increasing the 'area' presented to the PEs by each bit plane. To achieve a theoretical rate of 1000 MOPS we have to progress to DPA9 or to array of smaller DPA's, say 64 DPA6's. In either case, routing overheads may be significant unless new data paths are introduced. The second alternative is attractive because the 64 ACU's can work independently to achieve a higher effective PE utilisation, and given suitable connection paths reconfiguration can be used to suit problem geometry or avoid faulty DPA's. Further research is needed in this area. A connection scheme using two orthogonal sets of 8 data busses is shown in Figure 12. Being time-multiplexed, a move takes at least eight times as long as it does inside the DPA; however, in a single operation there is the choice of 1,65,... or 449 column or row steps. The use of direct memory access to the slow local stores would allow routing to be overlapped with computation. Within this general framework any of the DPA's may be replaced by a conventional processor, a large capacity store, or an I-O channel controller.

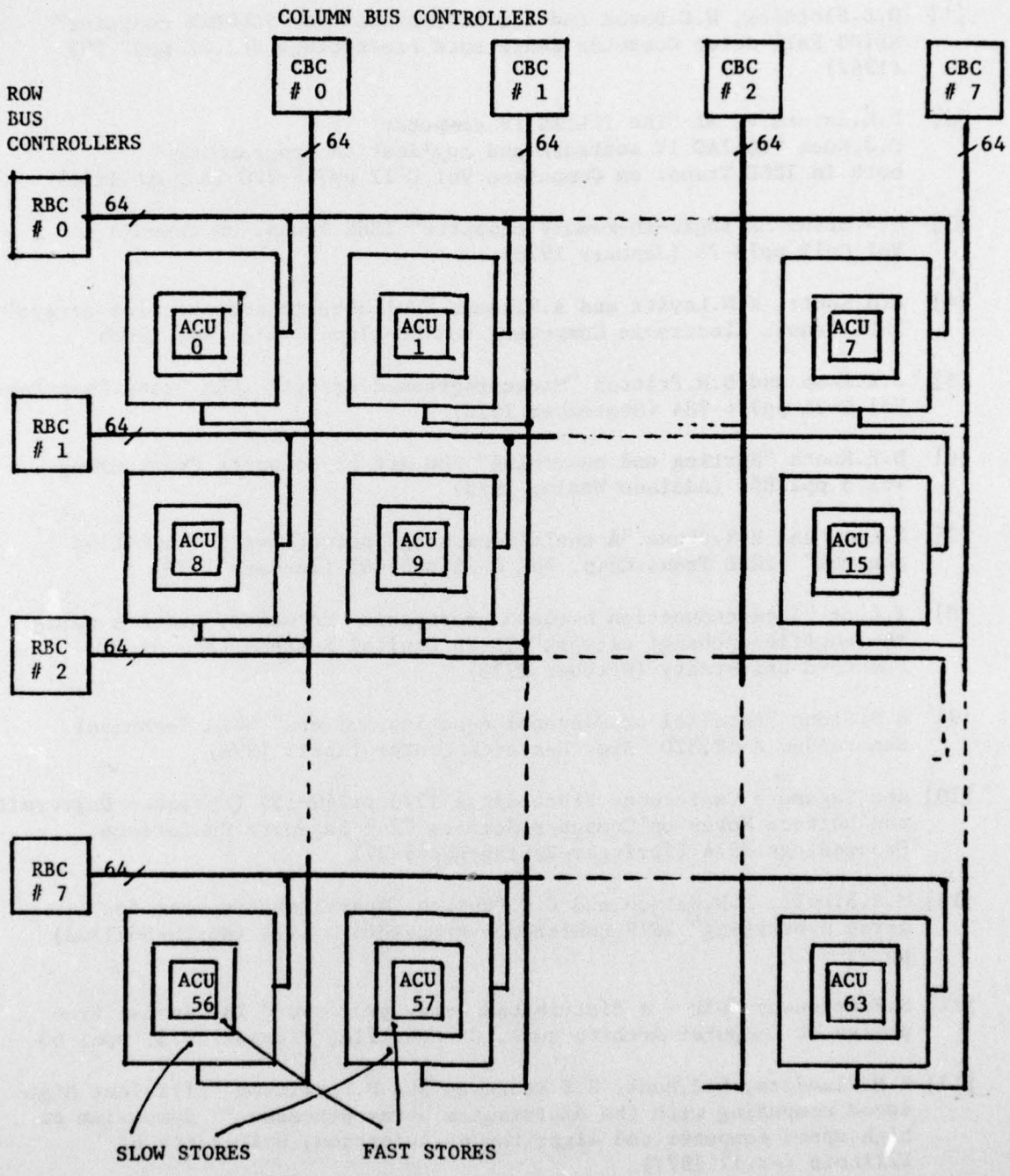


Figure 12: An array of DPA6's

REFERENCES

[1] D.L.Slotnick, W.C.Borck and R.C.McReynolds "The SOLOMON computer" AFIPS Fall Joint Computer Conference Proceedings Vol.22 pp97-107 (1962)

[2] G.H.Barnes et al "The ILLIAC IV computer"
D.J.Kuck "ILLIAC IV software and application programming"
both in IEEE Trans. on Computers Vol C-17 pp746-770 (August 1968)

[3] H.S.Stone "A logic-in-memory computer" IEEE Trans. on Computers Vol C-19 pp73-78 (January 1970)

[4] W.H.Kautz, K.N.Levitt and A.Waksman "Cellular interconnection arrays" IEEE Trans. Electronic Computers Vol EC-17pp443-451 (May 1968)

[5] J.R.Jump and D.R.Fritsch "Microprogrammed arrays" IEEE Trans.Computers Vol C-21 pp974-984 (September 1972)

[6] D.E.Knuth "Sorting and searching" The Art of Computer Programming Vol.3 pp228ff (Addison Wesley 1973)

[7] T.Lang and H.S.Stone "A shuffle-exchange network with simplified control" IEEE Trans.Comp. Vol C-25 pp55-65 (January 1976)

[8] T.Lang "Interconnection between processors and memory modules using the shuffle-exchange network" TR-76 Digital Systems Laboratory Stanford University (October 1973)

[9] H.S.Stone "Parallel tridiagonal equation solvers" NASA Technical Memorandum X-62,370 Ames Research Center (April 1974)

[10] see Sagamore Conference Proceedings 1973 pp140-159 (Syracuse University) and Lecture Notes on Computer Science #24: Sagamore Conference Proceedings 1974 (Springer-Verlag)pp209-271

[11] M.J.B.Duff, D.M.Watson and E.S.Deutsch "Aparallel computer for array processing" IFIP Conference Proceedings 1974 (North-Holland) pp 94-97

[12] S.F.Reddaway "DAP - a distributed array processor" 1st Annual Symposium on Computer Architecture, Gainesville, Florida 1973, pp61-65

[13] P.M.Flanders, D.J.Hunt, S.F.Reddaway and D.Parkinson "Efficient high speed computing with the distributed array processor" Symposium on high speed computer and algorithm organisation, University of Illinois (April 1977)

(14) DSL-TN-117

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: Technical Note No. 117
2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle): (6) COMPUTING IN STORE.
5. TYPE OF REPORT & PERIOD COVERED: (9) Technical Note
6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s): (10) John K. Iliffe
8. CONTRACT OR GRANT NUMBER(s): (15) N00014-75-C-0601

9. PERFORMING ORGANIZATION NAME AND ADDRESS:
Stanford Electronics Laboratories
Stanford University
Stanford, CA 94305

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS:
Office of Naval Research
Department of the Navy
Washington, DC 22217

12. REPORT DATE: (11) Jun 77
13. NUMBER OF PAGES: 55 (12) 57p.

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)

15. SECURITY CLASS. (of this report): Unclassified
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Reproduction in whole or in part is permitted for any purpose of the United States Government

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

408 071

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

These notes provide an introduction to the class of single-instruction, multiple-data stream computers with the simplest processing elements. Design principles are explained in terms of hypothetical Distributed Processor Arrays, with examples drawn from experimental systems. Emphasis is placed on; (a) minimising the cost differential when the DPA is compared with conventional main storage, and (b) designing the array control unit to support advanced forms of protection and language implementation. The influence of the DPA on general system design is examined briefly.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

JSEP REPORTS DISTRIBUTION LIST

	No. of Copies
Department of Defense	
Defense Documentation Center Attn: DDC-TCA (Mrs. V.Caponio) Cameron Station Alexandria, Virginia 22314	12
Asst. Dir., Electronics and Computer Sciences Office of Director of Defense Research and Engineering The Pentagon Washington, D.C. 20315	1
Office of Director of Defense Research and Engineering Information Office Lib. Branch The Pentagon Washington, D.C. 20301	1
ODDR&E Advisory Group on Electron Devices 201 Varick Street New York, New York 10014	1
Chief, R&D Division (340) Defense Dommunications Agency Washington, D.C. 20301	1
Director, Nat. Security Agency Fort George G. Meade Maryland 20755 Attn: Dr. T.J.Beahn	1
Institute for Defense Analysis Science and Technology Division 400 Army-Navy Drive Arlington, Virginia 22202	1
Dr. Stickley Defense Advanced Research Projects Agency Attn: Technical Library 1400 Wilson Boulevard Arlington, Virginia 22209	1
Dr. R. Reynolds Defense Advanced Research Projects Agency Attn: Technical Library 1400 Wilson Boulevard Arlington, Virginia 22209	1
Department of the Air Force	
AF/RDPS The Pentagon Washington, D.C. 20330	1
AFSC (LJ/ Mr. Irving R. Mirman) Andrews Air Force Base Washington, D.C. 20334	1
Directorate of Electronics and Weapons HQ AFSC/DLC Andrews AFB, Maryland 20334	1
Directorate of Science HQ AFSC/DLS Andrews Air Force Base Washington, D.C. 20334	1
LTC J.W. Gregory AF Member, TAC Air Force Office of Scientific Research Bolling Air Force Base Washington, D.C. 20332	5
Mr. Carl Sletten RADC/ETE Hanscom AFB, Maryland 01731	1
Dr. Richard Picard RADC/ETSL Hanscom AFB, Maryland 01731	1
Mr. Robert Barrett RADC/ETS Hanscom AFB, Maryland 01731	1

	No. of Copies
Dr. John N. Howard AFGL/CA Hanscom AFB, Maryland 01731	1
Dr. Richard B. Mack RADC/ETER Hanscom AFB, Maryland 01731	1
Documents Library (TILD) Rome Air Development Center Griffiss AFB, New York 13441	1
Mr. H.E.Webb, Jr. (ISCP) Rome Air Development center Griffiss AFB, New York 13441	1
Mr. Murray Kesselman (ISCA) Rome Air Development Center Griffiss AFB, New York 13441	1
Mr. W. Edwards AFAL/TE Wright-Patterson AFB Ohio 45433	1
Mr. R.D.Larson AFAL/DHR Wright-Patterson AFB Ohio 45433	1
Howard H.Steenbergen AFAL/DHE Wright-Patterson AFB Ohio 45433	1
Chief Scientist AFAL/CA Wright-Patterson AFB Ohio 45433	1
HQ ESD (DRI/Stop22) Hanscom AFB, Maryland 01731	1
Professor R.E.Fontana Head, Dept. of Electrical Engr. AFIT/ENE Wright-Patterson AFB Ohio 45433	1

	No. of Copies
Mr. John Mott-Smith HQ ESD (AFSC) MCIT - Stop 36 Hanscom AFB, MA. 01731	1
LTC Richard J.Gowen Professor Dept. of Electrical Engineering USAF Academy, Colorado 80840	1
AUL/LSE-9663 Maxwell AFB, Alabama 36112	1
AFETR Technical Library P.O. Box 4608, MU 5650 Patrick AFB, Florida 32542	1
ADTC (DLOSL) Eglin AFB, Florida 32542	1
HQ AMD (RDR/Col. Godden) Brooks AFB, Texas 78235	1
USAF European Office of Aerospace Research Technical Information Office Box 14, FPO, New York 09510	1
Dr. Carl E. Baum AFWL (ES) Kirtland AFB, New Mexico 87117	1
ASAFSAM/RAL Brooks AFB, Texas	1
<u>Department of the Army</u>	
HQDA (DAMAOARZ-A) Washington, D.C. 20310	1
Commander U.S. Army Security Agency Attn: IARD-T Arlington Hall Station Arlington, Virginia 22212	1

	No. of Copies
Commander U.S. Army Materiel Dev. & Readiness Command Attn: Tech. Library Rm 7S 35 5001 Eisenhower Ave. Alexandria, Virginia 22333	1
Commander Research Laboratory ATTN. DRXRD-BAD U.S. Army Ballistics Aberdeen Proving Ground Aberdeen, Maryland 21005	1
Commander Picatinny Arsenal Dover, New Jersey 07081 ATTN: SMUPA-TS-T-S	1
ATTN: Dr. Herman Robl U.S. Army Research Office P.O. Box 12211 Research Triangle Park North Carolina 27709	1
ATTN: MR. Richard O. Ulsh U.S. Army Research Office P.O. Box 12211 Research Triangle Park North Carolina 27709	1
Mr. George C. White, Jr. Deputy Director Pitman-Dunn Laboratory Frankford Arsenal Philadelphia, Penna. 19137	1
Commander Attn: Chief, Document Section U.S. Army Missile Command Redstone Arsenal, Alabama 35809	1
Commander U.S. Army Missile Command Attn: DRSMI-RR Redstone Arsenal, Alabama 35809	1
Commander Chief, Materials Sciences Division, Bldg. 292 Army Materials and Mechanics Research Center Watertown, Massachusetts 02172	

	No. of Copies
Commamder Harry Diamond Laboratories ATTN: Mr. John E. Rosenberg 2800 Posder Mill Road Adelphi, Maryland 20783	1
Commandant U.S. Army Air Defense School Attn: ATSAD-T-CSM Fort Bliss, Texas 79916	1
Commandant U.S. Army Command and General Staff College Attn: Acquisition, Library Div Fort Leavenworth, Kansas 66027	1
Dr. Hans K. Ziegler (AMSEL-TL-D) Army Member, TAC/JSEP U.S. Army Electronics Command (DRSEL-TL-D) Fort Monmouth, New Jersey 07703	1
Mr. J.E. Teti (AMSEL-TL-DT) Executive Secretary, TAC/JSEP U.S. Army Electronics Command (DRSEL-TL-DT) Fort Monmouth, New Jersey 07703	3
Director Night Vision Laboratory, ECOM ATTN: DRSEL-NV-D Fort Belvoir, Virginia 22060	1
Commander/Director Atmospheric Sciences Laboratory (ECOM) Attn: DRSEL-BL/DD White Sansa Missile Range New Mexico 88002	1
Director Electronic Warfare Lab., ECOM Attn: DRSEL-WL-MY White Sands Missile Range New Mexico 88002	1
Commander US Army Armament Command Attn: DRSAR-RD Rock Island, Illinois 61201	1

	No. of Copies
Project Manager Ballistic Missile Defense Program Office Attn: DACS-BMP (Mr. A. Gold) 1300 Wilson Blvd. Washington, D.C. 22209	1
Director, Division of Neuropsychiatry Walter Reed Army Institute of Research Washington, D.C. 20012	1
Commander, USASATCOM Fort Monmouth, New Jersey 07703	1
Commander, U.S. Army Communications Command Attn: Director, Advanced Concepts Office Fort Huachuca, Arizona 85613	1
Project Manager, ARTADS EAI Building West Long Branch, N.J. 07764	1
U.S. Army White Sands Missile Range STEWS-ID-R Attn: Commander White Sands Missle Range New Mexico 88002	1
Mr. William T. Kawai U.S. Army R&D Group (Far East) APO, San Francisco, Ca. 96343	1
Director, TRI-TAC Attn: TT-AD (Mrs. Briller) Fort Monmouth, N.J. 07703	1
Commander U.S. Army Electronics Command	1
Fort Monmouth, N.J. 07703	
Attn: AMSEL-RD-O (Dr. W.S. McAfee)	1
CT-L (Dr. G. Buser)	1
NL-O (Dr. H.S. Bennett)	1
NL-T (Mr. R. Kulinyi)	1
TL-B	1
VL-D	1
WL-D	1
TL-MM (Mr. Lipetz)	1
(cont'd)	

	No. of Copies
NL-H Dr. F. Schwering	1
TL-E Dr. S. Kronenberg	1
TL-E Dr. J. Kohn	1
TL-I Dr. C. Thornton	1
NL-B Dr. S. Amorsos	1
Col. Robt. W. Noce Senior Standardization Rep. U.S. Army Standardization Group, Canada Canadian Force Headquarters Ottawa, Ontario, Canada KIA OK2	1
Commander CCOPS-PD Fort Huachuca, Arizona 85613 Attn: H.A. Lasitter	
Department of the Navy	
Dr. Sam Koslov ASN (R&D) Room 4E741 The Pentagon Washington, D.C. 20350	1
Office of Naval Research 800 N. Quincy Street Arlington, Virginia 22217 Attn: Codes 100 102 201 220 221 401 420 421 427 (All Hands) 432 437	1
Naval Research Laboratory 4555 Overlook Aven. SW Washington, D.C. 20375 Attn: Codes 4000 - Dr. A Berman 4105 - Dr. S. Teitler 4207 - Dr. J. McCaffrey 5000 - Dr. H. North 5200 - Mr. A. Brodzinsky 5203 - Dr. L. Young 5210 - Dr. J. Davey	

	No. of Copies
Naval Research Laboratory 4555 Overlook Ave. SW Washington, D.C. 20375 Attn: Codes 5220 - Mr. H. Lessoff 5230 - Dr. R. Green 5250 - Cf. L. Whicker 5260 - Dr. D. Barbe 5270 - Dr. B. McCombe 5300 - Dr. M. Skolnik 5403 - Dr. J. Shore 5464/5410 - Dr. J. Davis 5500 - Dr. T. Jacobs 5509 - Dr. T. Giallorenzi 5510 - Dr. W. Faust 6400 - Dr. C. Klick 7701 - Mr. J. Brown	1
Director Office of Naval Research 495 Summer Street Boston, Mass. 02210	1
Director Office of Naval Research New York Area Office 715 Broadway 5th Floor New York, New York 10003	1
Director of Naval Research Branch Office 536 South Clark Street Chicago, Illinois 60605	1
Director of Naval Research Branch Office 1030 East Green Street Pasadena, Calif. 91101	1
Office of Naval Research San Francisco Area Office 760 Market St. Room 447 San Francisco, Calif. 94102	1
Harris B. Stone Office of Research, Development, Test & Evaluation NOP-987 The Pentagon, Room 5D760 Washington, D.C. 20350	1
Dr. A.L. Slafkosky Code RD-1 Headquarters Marine Corps Washington, D.C. 20380	1
R.N. Keeler NAVMAT - Code 03T CP # 5 2211 Jefferson Davis Hwy. Arlington, Virginia 20360	1
Mel Nunn NVMAT 0343 CP# 5, Room 1044 2211 Jefferson Davis Hwy. Arlington, Virginia 20360	1
Dr. F.I. Tanczos NAVAIR-03B JP# 1, Room 412 1411 Jefferson Davis Hwy Arlington, Virginia 20360	1
Dr. H.J. Mueller Naval Air Systems Command Code 310 JP # 1 1411 Jefferson Davis Hwy. Arlington, Virginia 20360	1
Mr. N. Butler Naval Electronics Systems Command Code 304 NC # 1 2511 Jefferson Davis Hwy. Arlington, Virginia 20360	1
Mr. L.W. Sumney Naval Electronics Systems Command NC # 1 2511 Jefferson Davis Hwy. Arlington, Virginia 20360	1
J.H. Huth NAVSEA - Code 03C NC # 3, Room 11E08 2531 Jefferson Davis Hwy. Arlington, Virginia 20362	1
Capt. R.B. Meeks Naval Sea Systems Command NC #3 2531 Jefferson Davis Hwy. Arlington, Virginia 20362	1

	No. of Copies
Naval Surface Weapons Center White Oak Silver Spring, Maryland 20910 Attn: Codes WR - 04 - W. Scanlon WR - 30 - Dr. J. Dixon WR - 303 - Dr. R. Allgaier WR - 34 - H.R. Riedl WR - 43 - P. Wessel	1
Naval Surface Weapons Center Dahlgren, Virginia 22448 Attn: Codes DF - J. Mills DF - 14 - K. Ferris DF - 36 - S. Leong	1
Naval Air Development Center Johnsville Warminister, Penna 18974 Attn: Codes 01 - Dr. R. Lobb 202 - T. Shopple 20212 - S. Campagna 2022 - G. Fer	1
Dr. Gernot M.R. Winkler Director, Time Service U.S. Naval Observatory Mass. Ave. at 34th St., N.W. Washington, D.C. 20390	1
Officer in Charge Carderock Laboratory David Taylor Naval Ship Research and Development Center Bethesda, Maryland 20034	1
Officer in Charge Annapolis Laboratory Naval Ship Research & Development Center Annapolis, Maryland 21402	1
Dr. G. Gould, Technical Director NavalCoastal Systems Laboratory Panama City, Florida 32401	1
M.J. Wynn Code 790 Naval Coastal Systems Laboratory Panama City, Florida 32401	1
Dr. W. A. VonWinkle Associate Technical Director for Technology Naval Underwater Systems Center New London, Connecticut 06320	1
Officer in Charge Naval Underwater Systems Center Newport, Rhode Island 02840	1
Dr. H.L. Blood Technical Director Naval Undersea Center San Diego, Calif. 95152	1
Dr. Robert R. Fossum Dean of Research Naval Postgraduate School Monterey, Calif. 93940	1
Naval Electronics Laboratory Center 271 Catalina Blvd. San Diego, Calif. 92152 Attn: Codes 0220 - H.T. Mortimer 2000 - P.C. Fletcher 2020 - V.E. Hildebrand 2100 - C.A. Nelson 2200 - J. Ritcher 2300 - C.W. Erickson 2400 - F.M. Tripak 2500 - W.E. Richards 3000 3400 - R. Coburn 4000 - C.E. Pierson 4600 - I. Lagnado 5000 - A.E. Beutel 5200 - R.R. Eyres 5300 - P.H. Johnson 5600 - W.J. Dejka	1
Naval Weapons Center China Lake, Calif. 93555 Attn: Codes 60 - Royce 601 - F.C.Essig 6013 - V.L. Rehn 6014 - D.J.White 6018 - J.M Bennett 6019 - N. Bottka (cont'd)	1

	No of Copies
Naval Weapons Center China Lake, Calif. 93555 Attn: Codes 605 - W.S. McEwan 5515 - M.H. Ritchie 3945 - D.G. McCauley 5525 - Webster 35 - D.J.Russell 55 - B.W. Hayes 3544 - H.W. Swinford 3815 - R.S. Hughes	1
D.E. Kirk Professor & Chairman, Electronic Engineering Sp-304 Naval Postgraduate School Monterey, Calif. 93940	1
Professor Sydney P. Parker Electrical Engineering Sp-62 Naval Postgraduate School Monterey, Calif. 93940	1
Dr. Roy F. Potter 3868 Talbot Street San Diego, Calif. 92106	1
Mr. J.C. French Electronics Technology Division National Bureau of Standards Washington, D.C. 20234	1
John L. Allen Deputy Director (Research & Advanced Technology) ODDR&E The Pentagon, Room 3E114 Washington, D.C. 20301	1
Leonard R. Weisberg Assistant Director (Electronics & Physical Sciences) ODDR&E The Pentagon Washington, D.C. 20301	1
George Gamota Staff Specialist for Research ODDR&E The Pentagon, Room 3D1079 Washington, D.C. 20301	1

	No. of Copies
Robert E. Frischell Johns Hopkins University Applied Physics Laboratory Laurel, Maryland 20810	1
Mr. G.H. Gleissmer Code 18 David Taylor Naval Ship R&D Center Bethesda, Maryland 20084	1
Commander Pacific Missile Test Center Code 4253-3 Point Mugu, Calif. 93042	1
Richard Holden DF - 34 Naval Surface Weapons Center Dahlgren Laboratory Dahlgren, Virginia 22448	1
Other Government Agencies	
Mr. F.C. Schwenk, RD-T National Aeronautics and Space Administration Washington, D.C. 20546	1
Los Alamos Scientific Lab Attn: Reports Library P.O. Box 1663 Los Alamos, New Mexico 87544	1
M. Zane Thornton Deputy Director, Institute for Computer Sciences & Technology National Bureau of Standards Washington, D.C. 20550	1
Director, Office of Postal Technology (R&D) U.S. Postal Service 11711 Parklawn Drive Rockville, Maryland 20852	1
NASA Lewis Research Center Attn: Library 21000 Brookpark Road Cleveland, Ohio 44135	1

	No. of Copies
Library - R51 Bureau of Standards Acquisition Boulder, Colorado 80302	1
MIT Lincoln Laboratory Attn: Library A-082 P.O. Box 73 Lexington, Mass. 02173	1
Dr. Jay Harris Program Director, Devices and Waves Program National Science Foundation 1800 G. Street Washington, D.C. 20550	1
Dr. Howard W. Etzel, Deputy Director Division of Materials Research National Science Foundation 1800 G. Street Washington, D.C. 20550	1
Dr. Dean Mitchell, Program Director Solid-State Physics Div. of Materials Research National Science	1

Non-Government Agencies

Director Research Lab. of Electronics Massachusetts Inst. of Tech. Cambridge, Mass. 02139	1
Director Microwave Research Institute Polytechnic Inst. of New York Long Island Graduate Center Route 110 Farmingdale, New York 11735	1
Assistant Director Microwave Research Institute Polytechnic Inst. of New York 333 Jay Street Brooklyn, New York 11201	

	No. of Copies
Director Columbia Radiation Laboratory Department of Physics Columbia University 538 West 120th Street New York, New York 10027	1
Director Electronics Research Laboratory University of California Berkeley, Calif. 94720	1
Director Electronics Sciences Laboratory University of Southern California Los Angeles, California 90007	1
Director Electronics Research Center The University of Texas at Austin Engineering-Science Bldg. 112 Austin, Texas 87812	1
Director of Laboratories Division of Engineering and Applied Physics - Tech. Reports Collection Harvard University Pierce Hall Cambridge, Massachusetts 02138	1